(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

THE SHORTEST PATH PROBLEM IN THE PLANE WITH OBSTACLES: A GRAPH MODELING APPROACH TO PRODUCING FINITE SEARCH LISTS OF HOMOTOPY CLASSES

by

Kevin D. Jenkins

June, 1991

Thesis Advisor: J. R. Thornton

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*) 53	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10 SOURCE OF FUNDING NUMBERS Program Element No / Project No / Task No / Work Unit Accession Number

11. TITLE (*Include Security Classification*)
THE SHORTEST PATH PROBLEM IN THE PLANE WITH OBSTACLES: A GRAPH MODELING APPROACH TO PRODUCING FINITE SEARCH LISTS OF HOMOTOPY CLASSES

12. PERSONAL AUTHOR(S) Kevin D. Jenkins

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*year, month, day*)	15. PAGE COUNT
Master's Thesis	From To	June 1991	126

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18. SUBJECT TERMS (*continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUBGROUP	Path Planning, Finding the Shortest Path in the Plane With Obstacles

19. ABSTRACT (*continue on reverse if necessary and identify by block number*)

The problem of finding the shortest path between two points in a plane containing obstacles is considered. The set of such paths is uncountably infinite, making an exhaustive search impossible. This difficulty is overcome by reducing the size of the search space. The search is first restricted to a countably infinite set by focusing attention on the set of homotopy classes. By applying simple optimality principles, a finite list of such classes is obtained whose union contains the shortest path. This process of simplification is accomplished by modeling the topology of the region with a graph. Optimality principles come into play during a graph traversal which is used to produce the finite search list. In addition, a computational investigation of two methods by which homotopy classes can be named is discussed, and properties of the graph models are investigated. The thesis of CPT Andre M. Cuerington, U.S. Army, calculates the actual shortest path using the search list produced here.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS REPORT ☐ DTIC USERS	Unclassified	
22a NAME OF RESPONSIBLE INDIVIDUAL J. R. Thornton	22b TELEPHONE (*Include Area code*) (408) 646-2741	22c OFFICE SYMBOL Ma/Th

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited.

The Shortest Path Problem in the Plane With Obstacles: A Graph Modeling Approach to Producing Finite Search Lists of Homotopy Classes

by

Kevin Dean Jenkins
Captain, United States Marine Corps
B.S., Virginia Military Institute, 1985

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN APPLIED MATHEMATICS

from the

NAVAL POSTGRADUATE SCHOOL
June, 1991

Author: _____
Kevin Dean Jenkins

Approved By: _____
John R. Thornton, Thesis Advisor

_____
Kim A. S. Query, Second Reader

_____
Harold M. Fredricksen, Chairman,
Department of Mathematics

ABSTRACT

The problem of finding the shortest path between two points in a plane containing obstacles is considered. The set of such paths is uncountably infinite, making an exhaustive search impossible. This difficulty is overcome by reducing the size of the search space. The search is first restricted to a countably infinite set by focusing attention on the set of homotopy classes. By applying simple optimality principles, a finite list of such classes is obtained whose union contains the shortest path. This process of simplification is accomplished by modeling the topology of the region with a graph. Optimality principles come into play during a graph traversal which is used to produce the finite search list. In addition, a computational investigation of two methods by which homotopy classes can be named is discussed, and properties of the graph models are investigated. The thesis of CPT André M. Cuerington, U.S. Army, calculates the actual shortest path using the search list produced here.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

TABLE OF CONTENTS

ACKNOWLEDGMENT

In the course of researching and preparing this thesis, countless hours were spent with Professor John Thornton. Without his continuous guidance and assistance this paper would not have been possible. In addition to his technical assistance, Dr. Thornton has established topological proofs which support the theories behind the mechanics of Chapter II and Appendix A. Further information regarding these proofs may be obtained from him through the Naval Postgraduate School Mathematics Department.

Much of the research for this project was conducted in conjunction with the thesis research of CPT André M. Cuerington, U.S. Army. Many months were spent with him in writing and debugging the FORTRAN program which is presented in Appendix A. Following the joint research and development of the shortest path problem, Chapter II and the text of Appendix A, with the exception of only minor changes, were written by CPT Cuerington for inclusion in this thesis.

I am gratefully indebted to both of these professionals. To both, I can only offer a heartfelt 'Thank you'.

I. INTRODUCTION

The industrial community relies today, in increasing measure, upon robots to handle a multitude of tasks. One goal is to develop robots, and fixed robot arms, with the ability to roam freely among obstacles. In this regard, the calculation of shortest paths is of obvious importance.

A. THE PROBLEM

As robot arms move, they must vary their configurations to position their end-effector or "hand". Once obstacles are introduced, the freedom of movement of the manipulator may become drastically reduced. The problem considered here is one of finding the shortest path from a starting point to a destination point in a planar region containing obstacles. This paper addresses several parts of the solution to this problem.

B. OVERVIEW OF THE SOLUTION

Between any two points in the plane, a path joining them may be chosen from an uncountably infinite set of alternatives. It is desirable to choose the shortest path from this set which, due to the presence of obstacles, may not be a straight line. An exhaustive search of the collection of possible paths is impossible, so another method must be developed.

The set of possible paths is partitioned into a collection of equivalence classes--mutually exclusive subsets which collectively exhaust the partitioned set. This partition is induced when an equivalence relation is defined on the set of possible paths. This set of equivalence classes is countably infinite.

Naming conventions are established which associate with each class a character string which allows the classes to be represented in a computer. Next, a finite list of candidate classes is produced to search.

A heuristic is then applied to this finite list of classes to place them in an order which saves computational effort.

The final step begins with this ordered list of candidate classes and solves the shortest path problem, class by class, in listed order. Savings of computational effort are realized when the class-by-class solution process can terminate without exhausting the ordered list of candidate classes.

The key ideas in the above overview are now considered in somewhat greater detail.

C. THE APPROACH

1. The Topology

Let P be the uncountably infinite set of all continuous, obstacle-avoiding paths from a starting point a to a destination point z. If p_i and p_j are <u>coterminal paths</u> in P, that is, paths which have the same starting point and

destination point, we say that p_i is homotopic to p_j if p_i can be mapped to p_j under a continuous function (with both endpoints fixed in place) without encroaching on any obstacles [Ref. 1:p. 223]. The homotopy relation is reflexive, symmetric and transitive and therefore defines an equivalence relation [Ref. 1:p. 223]. This relation induces a partition of P into a countably infinite collection of equivalence classes, known as homotopy classes [Ref. 1:p. 223].

2. Establishing the Naming Convention

In order to name homotopy classes, a reference frame is established to represent the topological relationships in the plane with obstacles. For a given path, p, a string of characters, R(p), is recorded which encodes information concerning the relationship of the path to obstacles.

Two algorithms are then presented which accept R(p) as input. These algorithms have the following important property which implicitly defines names for the homotopy classes: if p_i and p_j are coterminal paths and $R(p_i)$ and $R(p_j)$ are input to either of the two algorithms, then the outputs are identical if and only if p_i is homotopic to p_j [Ref. 2].

Appendix A presents a computational investigation which closely examines the methods employed by these two algorithms in producing the names for these homotopy classes.

3. Generation of Names for Candidate Equivalence Classes

An edge-labeled graph is constructed which models topological relationships within the region, where the nodes

represent subregions induced by the reference frame. Nodes are connected if their corresponding subregions are adjacent. The names of the desired homotopy classes may be obtained by traversing this graph. This paper presents algorithms both to construct the graph and to traverse it. This traversal produces a list of candidate classes, which contain all paths of minimal length.

The location of the origin affects the manner in which the plane is divided into wedges. In this regard, the graph which is created, given the location of the origin in one instance, may not be the same as the graph obtained if the origin is moved to a new point. This thesis presents a method to count the number of possible different graphs, given n obstacles, and a method to construct them.

4. Heuristic Ordering of Candidate Homotopy Classes

For each class on the list, a lower bound on the length of its shortest representative path is constructed. The list of classes is then arranged into increasing order of these bounds.

To obtain the bounds, a point is first chosen within each obstacle. A contraction deformation is then applied to each obstacle to "shrink" the obstacle so that the chosen point represents the obstacle.

Then some class is fixed and its shortest path is examined. As all obstacles are simultaneously contracted to their representing points, this shortest path has a limiting

position which is polygonal. The length of this polygonal path is the lower bound associated with that class. The length can be calculated from the class name and representing points without explicitly defining any contraction deformation.

5. Class by Class Solution of the Shortest Path Problem

In the final step of the solution, the classes on the ordered list of candidates are considered. The first class is removed from the list, i.e., that class with the smallest associated bound. The true shortest path in the named class is found by reversing the contraction previously applied to the obstacles, thereby transforming the polygonal path whose length provided the lower bound into the true shortest path. If the length of this path is smaller than the bound associated with the class on the top of the remaining list, the search is stopped. Otherwise, the first class is removed from that remaining list and the procedure is repeated. This procedure continues until the condition specified above is met, and the shortest path has been found.

D. SUMMARY

The solution to the problem of searching for the shortest path between two points in the plane with obstacles begins with consideration of a set of paths which is uncountably infinite. Through the homotopy relation, this set is reduced to a countably infinite set. The new set is further reduced

to a finite list by modeling the region with a graph and applying an optimality principle. This final list contains only those homotopy classes containing paths which are not self-crossing. From this list a shortest path is found.

Computational effort is further reduced through the use of a heuristic which orders the list of candidate classes by increasing order of their lower bounds. The heuristic used also facilitates a methodical search of the classes while solving the shortest path problem. The use of this heuristic does not, however, imply that the solution is approximate. The solution to the problem will be exact using this method.

E. THE CONTRIBUTION

Chapter II of this thesis presents the methods used to establish a reference frame given the plane with obstacles and an algorithm which is used to generate the homotopy class name of a given path.

Chapter III introduces a graph which is used to model the region. An algorithm is then presented which is used to traverse this graph to produce the list of candidate homotopy classes. Using the results of this algorithm, the thesis submitted by CPT André M. Cuerington, U.S. Army [Ref. 3], shows how to find the solution to the shortest path problem.

Chapter IV introduces some properties of the graph which are used to model the region.

Chapter V employs counting methods to determine the number of non-isomorphic graphs which may be encountered given a plane with some number of fixed obstacles. The number of non-isomorphic graphs represents the number of truly different reference frames which may be constructed in the region, thus producing different sets of names for the homotopy classes.

Finally, Appendix A presents a computational investigation which addresses the question of whether or not two algorithms used to name homotopy classes actually produce the same results.

II. NAMING EQUIVALENCE CLASSES

A. INTRODUCTION

In this chapter, a labeling scheme is established to represent the equivalence classes. This notation is used throughout this paper to organize the computational search for the shortest path.

An algorithm is presented that from a character representation of a path, p, names the path's respective equivalence class. It can be shown that, after being processed by the algorithm, different character representations of paths p_i and p_j yield the same output exactly when the two paths are in the same equivalence class [Ref. 2]. Thus, the name of a class will be the string obtained by applying the algorithm to any representative of the class.

The computation of class names depends on a reference frame, which in turn depends on a collection of obstacles. Although more than one reference frame can be drawn for a particular collection of obstacles, the choice of a particular frame fixes the representation of all homotopy classes.

Once a reference frame is developed, the name of the homotopy class for a path can be determined by a two-step procedure. First, a character string R(p) is calculated which encodes certain information about the path taken through the

obstacles. Second, this character string is accepted as input to an algorithm, which then produces a name for the equivalence class to which the particular path belongs. This algorithm produces the class names in terms of the same alphabet used to create the initial character string. The results from the algorithm described, Algorithm 1, will be used throughout this analysis as the class name associated with a given path.

B. CONSTRUCTION OF REFERENCE FRAME

Let b_k be an arbitrary point in obstacle B_k, $k = 1,2,\ldots,n$, where n is the number of obstacles in the region. A point c is chosen and a line drawn through each b_k, infinite in extent in each direction, and having the following properties: there is an open disk, δ, centered at c such that $\delta \cap B_k = \emptyset$ for all $k = 1,\ldots,n$, and the n lines connecting c with the points b_k are distinct. Such a point c can always be found as the above two conditions are satisfied by any point in the planar region that is neither on an obstacle nor on the $n(n-1)/2$ lines determined by pairwise choices of distinct b_k.

To draw a reference frame, n lines are first constructed joining c with each b_k. The line from c to b_k is labeled as L_k. Each line is then partitioned into two directed, semi-infinite rays and one finite length line segment. The ray emanating from c in the direction away from b_k is called α_k. The ray emanating from b_k and away from c is denoted β_k. The

remaining line segment, $[c,b_k]$, is also denoted α_k. The reference frame is the collection of line segments and rays so constructed, as illustrated in Figure 2.1.

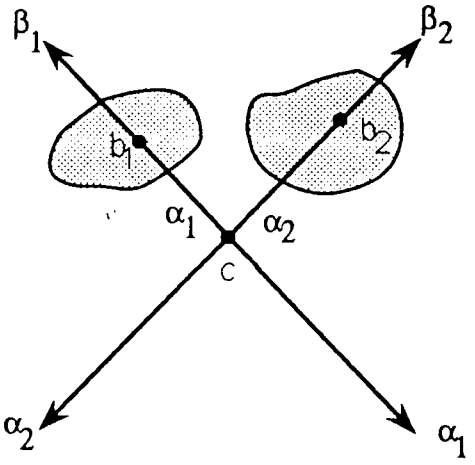


Figure 2.1 A Reference Frame With n = 2 Obstacles

C. RAW CHARACTER STRINGS

A reference frame is fixed in a region T. Let a and z be points in T, and p be a directed path in T from a to z. Then the raw string of p, denoted R(p), is defined to be the ordered sequence of characters obtained by following p from a to z and recording, in order, the names of the rays that are crossed.

Two special cases must be addressed to make the above definition complete. First, in the case that p crosses no reference rays, we set R(p) = e, where e denotes the empty string. Second, if p crosses through c (simultaneously

crossing all α_j, the names of all α_j will be recorded in order of increasing subscript.

Thus, raw strings are of the form

$$R(p) = x_1 x_2 \ldots x_m$$

where each x_j belongs to the alphabet

$$A = \{e, \alpha_1, \alpha_2, \ldots, \alpha_n, \beta_1, \ldots, \beta_n\}.$$

The character x_j ($j = 1, 2, \ldots, m$) is the name of the j^{th} reference ray crossed by p. Figure 2.2 shows a pair of paths connecting the two points a and z in the region with two obstacles.

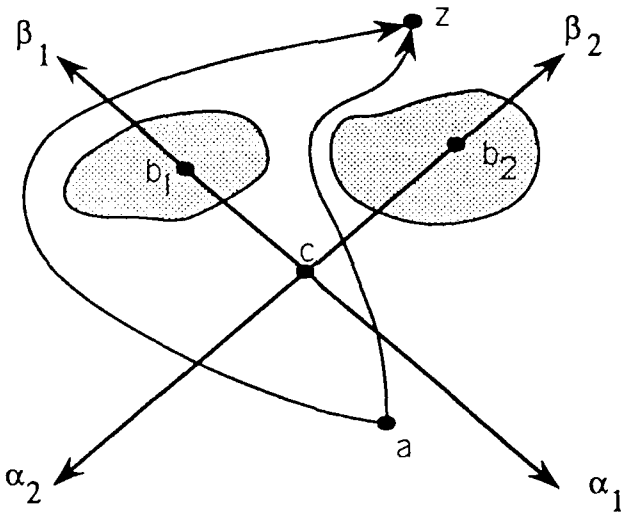


Figure 2.2 Paths $\alpha_2\beta_1$ and $\alpha_1\alpha_2$

D. ALGORITHM 1

Algorithm 1 accepts as input a raw string R(p) for any path p in T and any reference frame. The output is a string, denoted C(R(p)), of characters also chosen from the

alphabet A. This output string C(R(p)) is the name of the homotopy class to which p belongs.

The algorithm is presented in terms of two functions. The first is the sorting function σ. Let $S = x_1 \ldots x_m$ be any string over A. If S contains a two character substring $x_k x_{k+1} = \alpha_j \alpha_i$ with $i<j$, then $\sigma(S)$ is the string which results by reversing the order of the leftmost such substring. Figure 2.3 depicts two such strings and makes clear that such paths are homotopic. If S contains no such two character substring, then $\sigma(S) = S$.

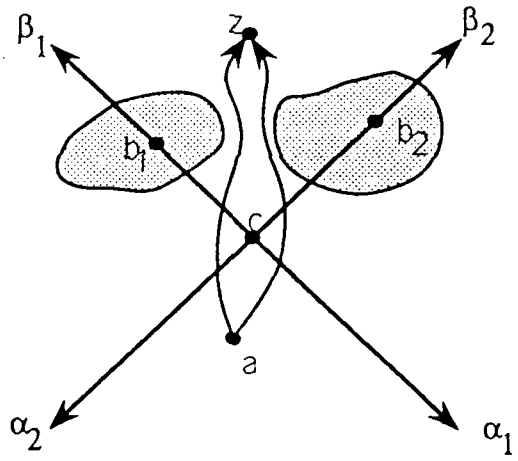


Figure 2.3 Homotopic Alpha Strings $\alpha_2\alpha_1$ and $\alpha_1\alpha_2$

Repeated application of σ sorts all substrings that consist entirely of α_j's into non-decreasing order of subscript. $\Sigma(S)$ is defined to be the string which results when this sorting is complete.

The second function is the cancellation function χ. If string S contains a two character substring $x_k x_{k+1}$ with $x_k = x_{k+1}$ then $\chi(S)$ is the string which results by removing the leftmost such two character substring. Otherwise $\chi(S) = S$. Such cancellation is intuitive, for if a reference ray is crossed twice consecutively, then that is equivalent to not crossing at all. Thus repeated application of χ cancels all pairs of adjacent like occurences of identical characters. Let $X(S)$ be defined as the string which results when all such possible cancellation is complete.

With these definitions complete, Algorithm 1 is given in Figure 2.4.

```
BEGIN
k ← 0
S_k ← R(p)
WHILE S_k not equal to X(Σ(S_k))
    k ← k+1
    S_k ← X(Σ(S_k-1))
END WHILE
C(R(p)) ← S_k
END
```

Figure 2.4 Algorithm 1

The output from Algorithm 1, C(R(p)), is called the canonical representation of p and is the unique name for the homotopy class of p [Ref. 2]. That name is used throughout this paper.

III. A GRAPH MODEL FOR NAMING CLASSES

This chapter presents a graph used to model the topological relationships in the region. The graph is then used to produce a finite list of names of candidate homotopy classes for subsequent searching.

A. MODELING THE TOPOLOGICAL SPACE

The reference frame, once established, divides the region into 2n "wedges". Each wedge is represented by a node. Two nodes are connected by one edge if the wedges they represent share a boundary of positive length corresponding to only an α ray, or two edges if the two wedges share a boundary corresponding to both α and β rays. No edge connects two nodes if the wedges they represent have no common boundary other than the origin, c. Figure 3.1 illustrates this procedure of replacing the wedges by their associated graph with n = 2.

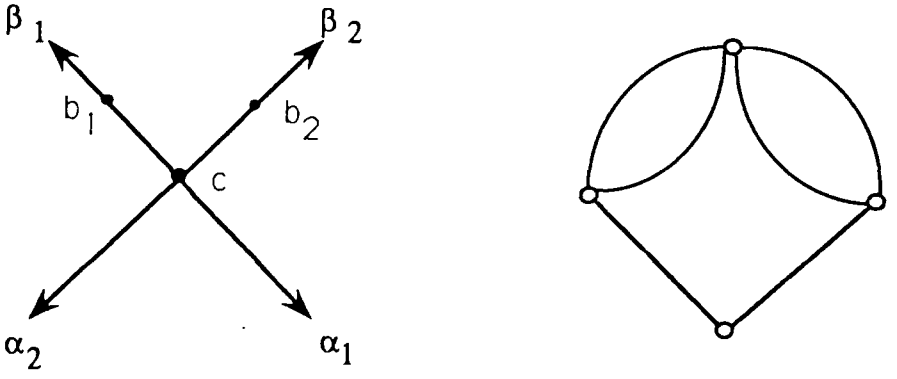


Figure 3.1 Establishing the Nodes and Edges of the Graph

Each edge of the graph is then labeled with an α_k or a β_k. An α_k is assigned to an edge if the incident nodes represent wedges sharing a boundary created by an α_k reference line. The β_k labels are similarly assigned. When parallel edges connect adjacent nodes the outermost edge is labeled with β_k, the innermost edge with α_k. This labeling coincides with the fact that the β_k ray for each obstacle is on the outboard side of the obstacle with respect to the origin, c. Figure 3.2 illustrates the edge labeling for n = 2.

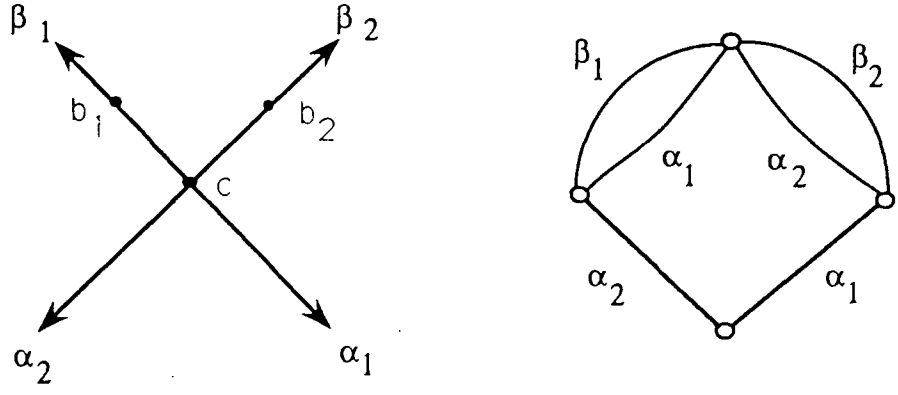


Figure 3.2 Edge Labeling for Graph with n = 2

In constructing the model, wedges and nodes are numbered counterclockwise using upper case Roman numerals.

We define a ring graph to be such a graph used to model a region. Figure 3.3 illustrates the region with n = 2 and its associated ring graph.

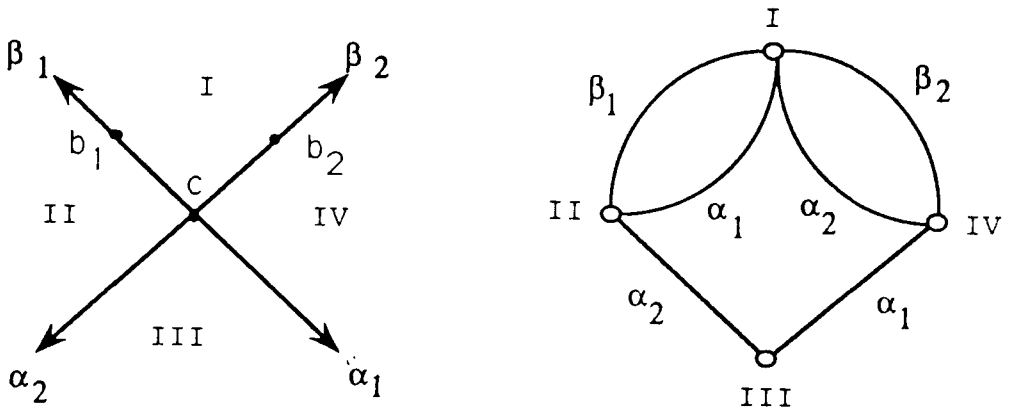
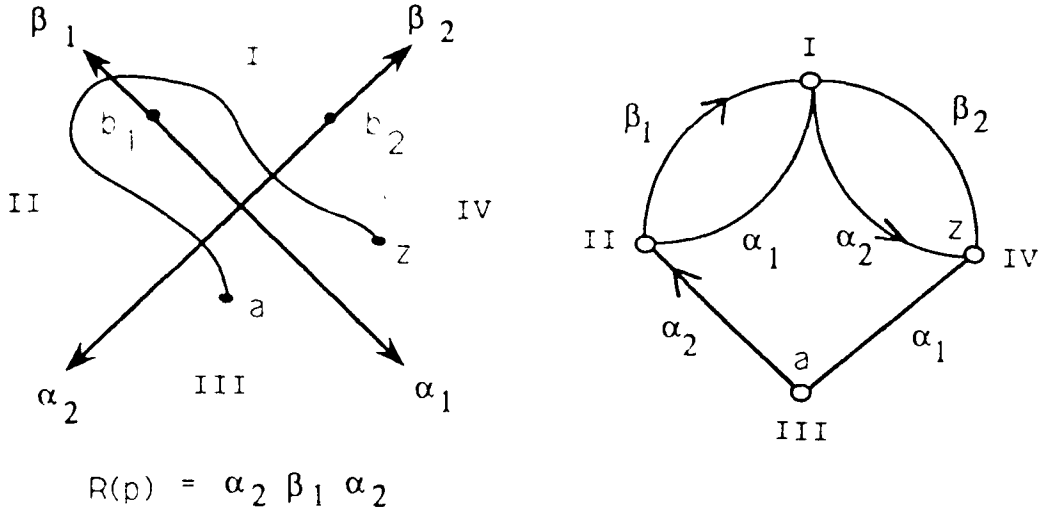


Figure 3.3 Region and Ring Graph for n = 2

If a and z are points in the topological space, T, and p is <u>any</u> path from a to z, then the string of characters, R(p), which represents that path can be produced by tracing p from a to z in T and recording R(p) as the list of rays crossed, as defined in Chapter II. However, R(p) can also be produced by traversing the ring graph and visiting the nodes which correspond to the wedges visited by p in T and recording the name of each edge as it is traversed. Figure 3.4 shows a path in a region where n = 2, and illustrates the path in the corresponding ring graph.

In constructing the model, it becomes unnecessary to generate a complete graph on the 2n nodes by eliminating the origin as a boundary between any two wedges. This may be justified by observing that any path passing directly through the origin could be perturbed in such a manner that any

analysis of the path would remain unchanged. This situation is illustrated in Figure 3.5.



$$R(p) = \alpha_2 \beta_1 \alpha_2$$

Figure 3.4 Path Represented by Ring Graph for n = 2

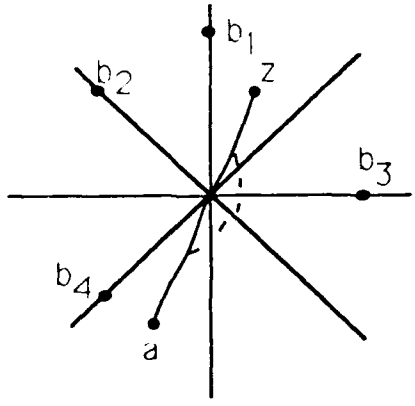


Figure 3.5 Perturbation of a Path Through the Origin

B. PRODUCING CLASS NAMES USING GRAPH TRAVERSAL

Any path p in T has a character string representation associated with it which may be obtained by traversing the graph. A list of character strings which name the homotopy classes is desired. This list of names is finite and includes only the names of those classes whose representative of shortest length is not self-crossing and starts and finishes in the appropriate wedges.

A list of character strings is constructed by beginning at some node and searching the graph using a modified Breadth First Search (BFS) algorithm. This search utilizes a first-in/first-out queue which stores the current list of character strings and an element which denotes the node in which each string terminates.

To illustrate the procedure, we begin with an empty string and the starting node of the graph on the queue. Then we copy the first string and the node denoting the current position from the queue. A new character which represents an edge which may be traversed in exiting the current node is added to the string. This new string is both printed and added to the bottom of the queue. All similar strings are generated by traversing <u>each</u> of the edges which may be traversed from the current node. The procedure is then repeated for the next string on the queue. Figure 3.6 presents this search in algorithmic language.

```
k := bottom of queue
Q := top of queue
v() := vertex on queue
s() := string on queue

BEGIN
k ← 1
Q ← 1
v(k) ← start vertex
s(k) ← e
WHILE (Q≤k) DO
    string ← s(Q)
    vertex ← v(Q)
    FOR EACH EDGE OF THE FORM (vertex, vertex2) DO
        k ← k+1
        char ← label[(vertex, vertex2)]
        s(k) ← string:char
        v(k) ← vertex2
        OUTPUT s(k), v(k)
    Q ← Q+1
END WHILE
END
```

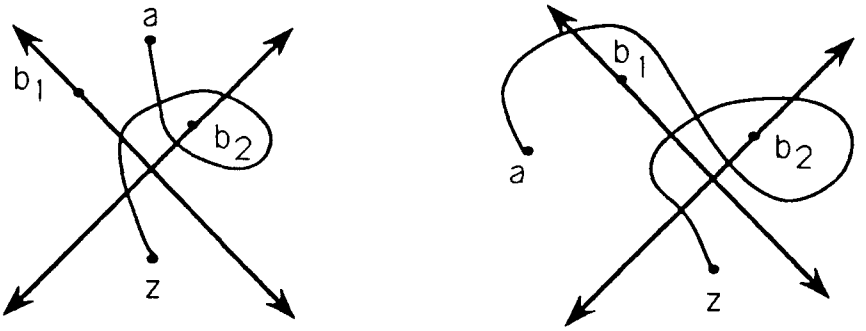
Figure 3.6 Modified BFS Algorithm

In the absence of some criterion for terminating this search, an ever-growing list would be produced. A new string would be produced for every node which is adjacent to the current node. It is easily seen that this infinite list would eventually include the character string representation of every possible path in T which starts in some given wedge. In order to accomplish the task of reducing this list to a finite list of names of candidate homotopy classes, some stopping criterion is required.

By examining the list of strings, it can be seen that some of the strings are not desirable because they are representative of classes which contain only paths which are

self-crossing or are subject to cancellation under Algorithm 1. These paths should be removed from the list. Since all of the extensions of these excluded strings are also undesirable, they too should be excluded. The algorithm then terminates after producing a finite list. That the list is finite is substantiated by using the Pumping Lemma of finite state machines. This lemma states that for any language, an infinite string is possible if and only if cycles are permitted [Ref. 4:p. 77]. The language here consists of the R(p) strings. A cycle in the language is representative of a loop around any obstacle or set of obstacles. Therefore, if we remove the cycles, a language with a finite number of words is produced, which in this case is a finite number of paths.

Two such undesirable strings are shown in Figure 3.7. These strings represent paths which either wrap around an obstacle or cross themselves by circling two obstacles in figure eight fashion.



$\alpha_2\beta_2\alpha_1\alpha_2$ $\beta_1\alpha_2\beta_2\alpha_1\alpha_2$

Figure 3.7 Paths Which Wrap and Cross

By performing several checks on each string as it is generated by the search algorithm, those strings which have an undesirable form may be "sieved" from the list. In other words, these strings will not be placed on the queue, and will therefore no longer be considered in the search. The criteria which are applied to each string to accomplish this sieving are given below.

The first criterion is that any string containing a substring of the form $\alpha_k, \beta_k, \alpha_k$ or $\beta_k, \alpha_k, \beta_k$ is not placed on the list. Such a string represents a class whose shortest representative wraps around an obstacle and is self crossing. The second criterion is that strings containing substrings of the form $\alpha_k, \beta_m, \alpha_m, \beta_k$ or $\beta_k, \alpha_m, \beta_m, \alpha_k$ represent classes whose shortest representative is also self crossing. They too are not placed on the list. Figure 3.8 provides examples of paths which produce substrings of these types, and also demonstrate how the shortest representative in each class crosses itself.

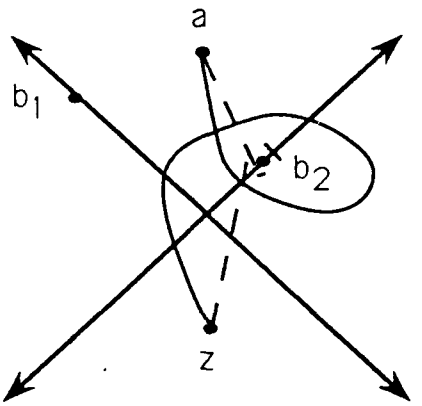
A third criterion eliminates from the list those strings which contain two like α's not separated by <u>any</u> β. Such strings are subject to cancellation as defined in Algorithm 1 presented in Chapter II. Once cancellation takes place the resulting string merely duplicates some string already on the list.

Another criterion introduces the restriction that the algorithm may not traverse an edge of the graph two times in immediate succession. Therefore, generation and cancellation

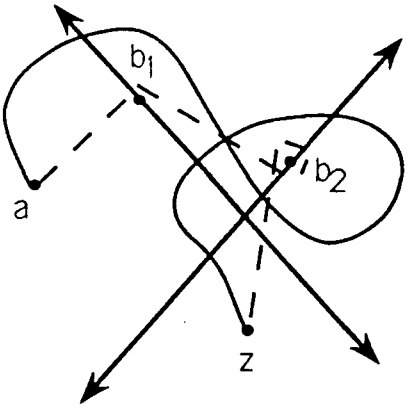
of such strings is not required. Also, duplication of strings can not occur.

Finally, those strings which do not terminate at the desired node are eliminated from the list.

What remains on the list upon termination of the search given these sieving rules are the names of only those candidate homotopy classes which are to be used in the subsequent search for a shortest path.



$\alpha_2\beta_2\alpha_1\alpha_2$



$\beta_1\alpha_2\beta_2\alpha_1\alpha_2$

Figure 3.8 Examples of Self Crossing Representatives

IV. PROPERTIES OF THE RING GRAPH

This chapter introduces some of the properties of the ring graphs obtained when the number of obstacles is greater than one. The case in which there is a single obstacle is addressed briefly at the end of the chapter.

As stated in Chapter II, each reference line, L_k, in the region containing n fixed obstacles is partitioned into three parts--two semi-infinite rays and one finite line segment. Since there is a one-to-one correspondence between these components and the edges of the ring graph, it is easily seen that the total number of edges in the graph will equal 3n for any $n \geq 2$.

Now, we superimpose a ring graph over its corresponding reference frame, as illustrated in Figure 4.1. We first observe that, by definition, each L_k contains exactly one point b_k. That point contributes to the existence of two edges connecting the adjacent wedges. Likewise, since no such b_k lies on L_k on the opposite side of c, only one edge connects the adjacent wedges. Therefore, it is easily seen that in every ring graph with $n \geq 2$, there are <u>always</u> two parallel edges opposite a single edge.

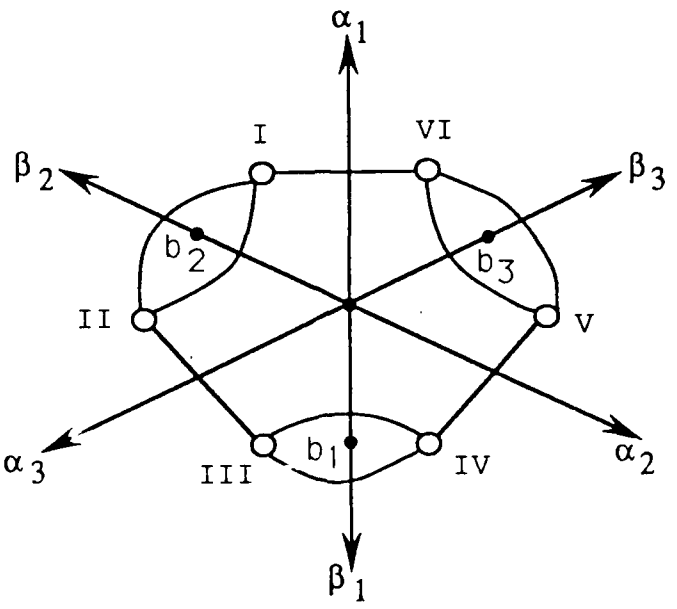


Figure 4.1 Region and Ring Graph Superimposed for n = 3

The number of edges incident with a node is called the degree of the node [Ref. 5:p. 86]. Consider the degree of the nodes of the ring graph. Every node in a given ring graph is of degree two, three or four. The sum of the degrees of the nodes in a graph is equal to twice the number of edges in that graph [Ref. 5:p. 86]. Since it has already been stated that the number of edges in the ring graph equals 3n, then for each ring graph,

$$\sum_{i=1}^{2n} degree(vertex\ i) = 6n \ . \qquad (4.1)$$

Since we are interested in the degree of incidence of individual nodes we rewrite (4.1) in the form

$$6n = \sum_{i=2}^{4} i(\textit{number of vertices of degree } i) \ . \qquad (4.2)$$

It is an easy combinatorial result that in any graph, there are an even number of nodes of odd degree [Ref. 5:p. 91]. Therefore, in any ring graph, the number of nodes of degree three is even.

Define g, h and j to be the number of nodes of degree two, three and four, respectively. Then the total number of nodes is

$$g + h + j = 2n \ , \qquad (4.3)$$

and (4.2) may be rewritten as

$$2g + 3h + 4j = 6n \ . \qquad (4.4)$$

Multiplying both sides of (4.3) by 3 and combining equations (4.3) and (4.4) yields

$$3(g + h + j) = 2g + 3h + 4j \ . \qquad (4.5)$$

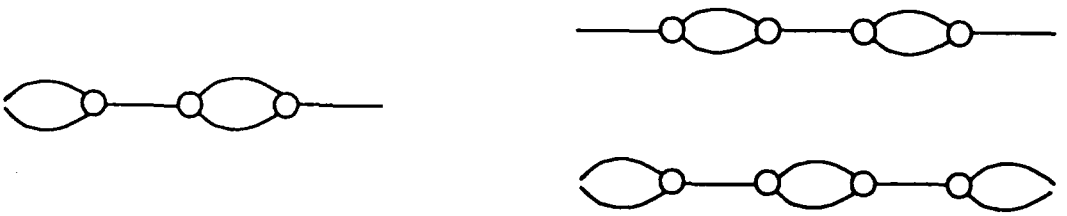
The solution to (4.5) shows that g = j or simply that the number of nodes of degree two equals the number of nodes of degree four in <u>any</u> ring graph with n ≥ 2. It may also be

verified by examining any ring graph that every node of degree two is directly opposite a node of degree four. Hence, there must be an equal number of each.

All cycles in ring graphs are on an even number of nodes. Therefore these graphs may be classified as being of bipartite type. That is, the set of nodes may be partitioned into two sets such that all edges of the graph connect a node in one set to a node in the other set [Ref 5:p. 240]. Since the nodes are labeled sequentially in a counterclockwise direction, the two sets in this partition consist of those nodes whose labels are even integers and those whose labels are odd integers, respectively.

This bipartite structure allows certain observations concerning the distance travelled in moving from node to node through the ring graph. Assigning an equal weight of one to each edge, the distance travelled along a path is simply the total number of edges traversed (with the multiplicity of traversals of individual edges included). It is easily seen that any path which begins and ends at nodes whose labels are of like parity will have even length. On the other hand, a path which begins and ends at nodes whose labels are of differing parity will have odd length.

Consider a chain of m nodes of degree three. For m odd, the chain appears as that shown in Figure 4.2(a). For m even, the chain takes on one of the forms shown in Figure 4.2(b).



(a) (b)

Figure 4.2 Chains of Nodes of Degree Three

Consider a chain such as the one illustrated in Figure 4.2(a). It is easily seen that in concatenating this chain with nodes of degree two and four, it is possible to do so with only a degree two node on one end and only a degree four node on the other. Figure 4.3 illustrates the extension of such a chain.

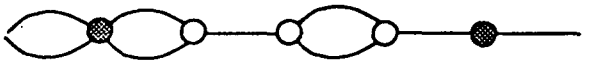


Figure 4.3 Concatenation of Even Degree Nodes

Using a similar construction, chains such as those illustrated in Figure 4.2(b) may be used to show, as seen in Figure 4.4, that only nodes of the same degree (two or four) may be concatenated on both ends.

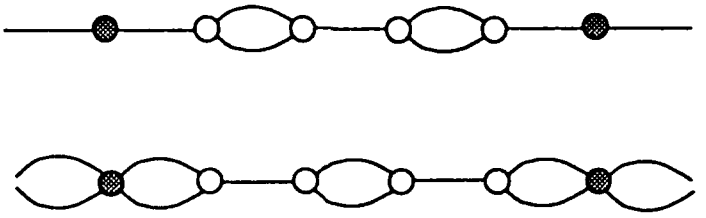


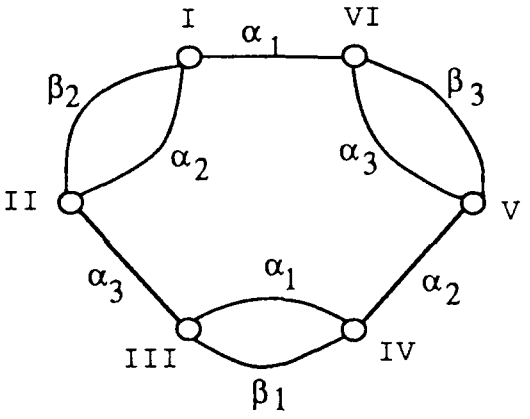
Figure 4.4 Concatenation of Even Degree Nodes

These facts lead to the observation that any string of nodes of degree two and any string of nodes of degree four must be separated by an odd number of nodes of degree three. Furthermore, if two nodes of degree two (or four) are separated by nodes of degree three, the number of degree three nodes must be even.

It is possible to represent a given ring graph by an adjacency matrix, A. The resulting matrix is a symmetric $2n \times 2n$ matrix in which the (i,j) element is a 1 if there is a single edge connecting nodes v_i and v_j, a 2 if two parallel edges connect the nodes, or a 0 if the nodes are not adjacent. Figure 4.5 shows a ring graph with n = 3 and its associated adjacency matrix. By virtue of the conventions used to label the nodes, it is observed that all nonzero entries in the adjacency matrix are on the superdiagonal, the subdiagonal and in the (2n,1) and (1,2n) positions. (All entries on the main diagonal of A are zero since no ring graph contains any edges which connect a node to itself.) Hence, the matrix becomes very sparse as n becomes large.

A basic property of an adjacency matrix is that the sum of the values in the i^{th} row (column) of the matrix is equal to the degree of node v_i in the graph [Ref. 5:p. 88].

As stated above, the conventions used to label the nodes of the ring graph indicate that any two nodes whose labels are of like parity must be an even distance apart. By the bipartite property of the graph, all (i,j) entries in A, where i and j are of like parity will necessarily be zero.



$$A = \begin{bmatrix} 0 & 2 & 0 & 0 & 0 & 1 \\ 2 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 2 \\ 1 & 0 & 0 & 0 & 2 & 0 \end{bmatrix}$$

Figure 4.5 A Ring Graph and its Adjacency Matrix, A

Another theorem from graph theory states that the number of paths of length m from node v_i to node v_j is the (i,j) entry of the m^{th} power of the adjacency matrix, A^m [Ref. 5:p. 144]. This theorem leads to the further conclusion that when m is odd, all (i,j) entries where i and j are of like parity will be zero. On the other hand, when m is even, all (i,j) entries

for i and j of opposite parity will be zero. Naturally, all entries not mentioned above may or may not be zero, and depend on the actual layout of the region and its corresponding ring graph.

In the case where there is only one obstacle, n = 1, a special case of the graph arises. Since one obstacle divides the region into two half planes, there are only two nodes in the graph with two parallel edges connecting them. Any two points in such a region may be connected by a path crossing either one α or one β edge. Therefore, this case is not considered in the analyses discussed in this paper.

V. NON-ISOMORPHIC RING GRAPHS

A. INTRODUCTION

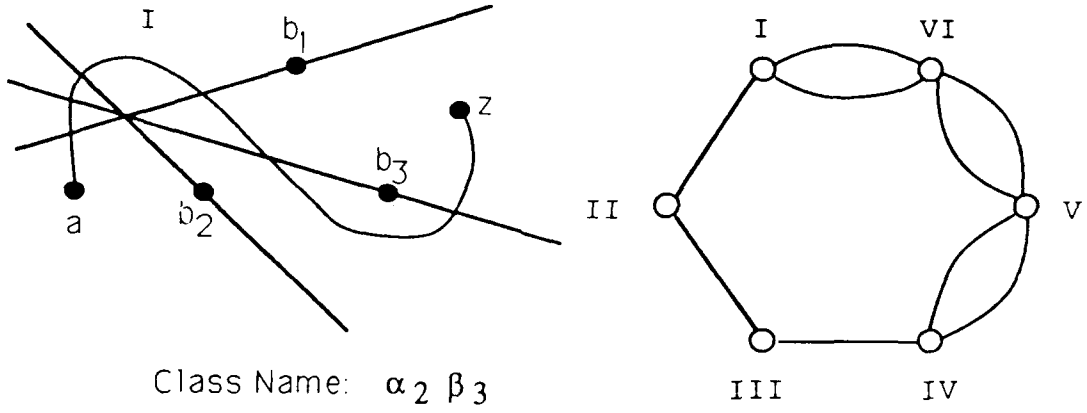
The geometry of a region is fixed by the location of the obstacles. However, by varying the location of the origin, different reference frames may be established. These differences may lead to different ring graphs and therefore different homotopy class names. Figure 5.1 illustrates this possibility for n = 3, where two different reference frames result in different ring graphs and different class names for the same path. This leads to the following question: **How many distinct ring graphs exist for a given collection of n fixed obstacles?**

It is shown in the sequel that the answer to the question may be found by determining the number of non-equivalent edge two-colorings of a 2n-gon subject to the restriction that opposite edges of the polygon are colored differently. Burnside's Lemma is used to determine this number.

It must be pointed out that coloring a graph in the context of this paper is <u>not</u> the traditionally accepted definition of a graph coloring. Here, adjacent edges (or later, vertices) <u>may</u> be colored the same color.

Once the number of different ring graphs has been determined, Polya's Pattern Inventory Theorem is used to construct them.

A complete example illustrating the procedures presented in this chapter is provided in Appendix C. Reference should be made to this example as new concepts are introduced.



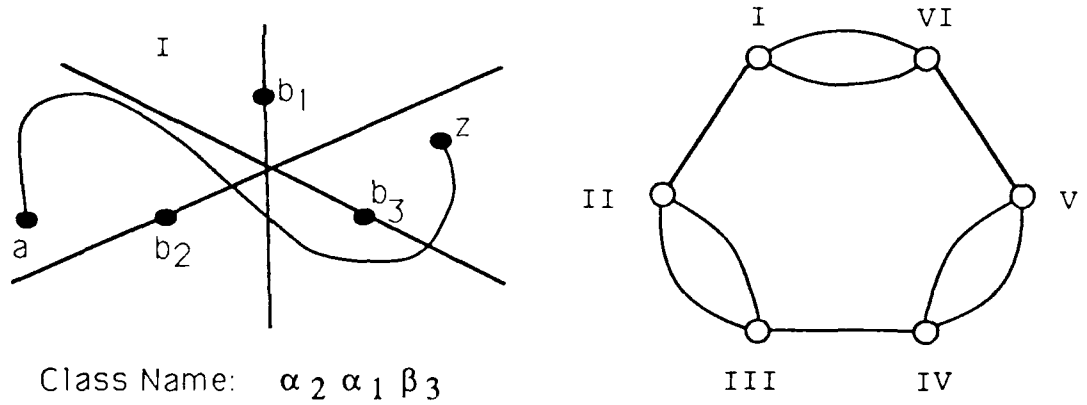


Figure 5.1 Effect of Reference Frame on Ring Graph and Class Names

B. THE RING GRAPH/POLYGON CONNECTION

"A graph is completely determined by specifying its node and edge sets."[Ref. 6:p. 3] Any two graphs are <u>isomorphic</u> if

there exists a one-to-one correspondence between their node sets which preserves adjacency [Ref. 6:p. 4]. The question at hand then becomes: How many non-isomorphic ring graphs exist for a given collection of n fixed obstacles?

Each ring graph associated with a problem containing n obstacles may be modeled using a 2n-gon whose vertices correspond to the nodes of the graph. The edges of the polygon are then colored as follows: a black edge (bold line) connects those vertices whose corresponding nodes are connected by two parallel edges; a white edge (fine line) connects those vertices whose corresponding nodes are connected by a single edge; and no edge connects those vertices which correspond to non-adjacent nodes. From this point forward, only single and double edges will be considered.

It has been established that, in a ring graph, each set of parallel edges is opposite a single edge. Accordingly, the corresponding restriction on the polygon is that opposite edges must be colored differently. Figure 5.2 shows a ring graph for n = 2 and its associated 2n-gon.

Since the configuration of a ring graph is defined by its node and edge sets, the location and/or spacing of its nodes is irrelevant. In constructing a graph then, the nodes may be thought of as being equally spaced around the ring. Therefore the 2n-gon associated with any ring graph may take the form of a regular polygon on 2n vertices.

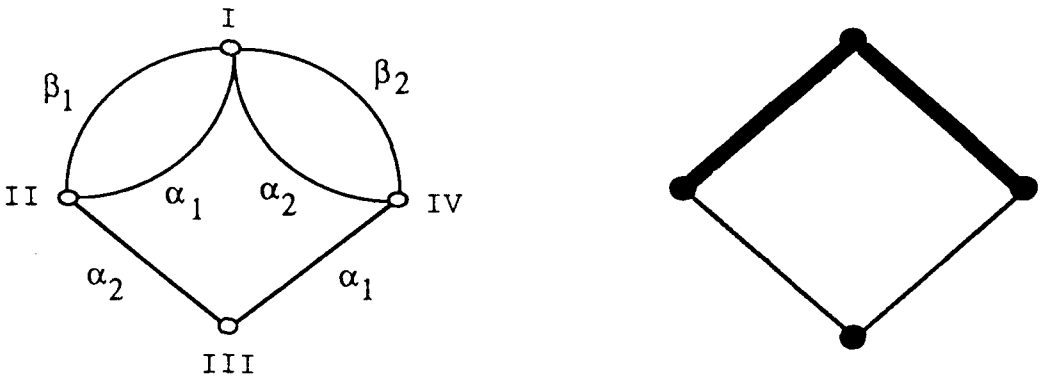


Figure 5.2 Ring Graph and 2n-gon for n = 2

With these ideas in mind, the problem of determining the number of non-isomorphic ring graphs given n fixed obstacles is the same as determining the number of non-equivalent edge two-colorings of a regular 2n-gon subject to the restriction that opposite edges are differently colored.

C. COUNTING NON-EQUIVALENT EDGE TWO-COLORINGS

1. Establishing the Permutations

By coloring one edge of the 2n-gon, the restrictions which have been established previously automatically fix the color of the opposite edge. Therefore, of the 2n edges, n of them may be colored independently leading to a total of 2^n possible colorings.

Any pair of two-colorings of the polygon are considered equivalent if there is a rigid motion of the 2n-gon that maps the polygon onto itself [Ref 7:p. 335], and which

permutes one coloring into the other. Permissible rigid motions, or permutations, will be limited to rotations about the center of the polygon, reflections about diagonals drawn through opposite vertices, and reflections about bisectors drawn through opposite edges. Figure 5.3 illustrates each of these permutations using the square generated by n = 2. Examination of any 2n-gon reveals that there are a total of 2n rotations, n reflections about diagonals and n reflections about bisectors. If G is defined to be the set of all permissible permutations on the set of edges of the 2n-gon, then it follows that $|G| = 4n$.

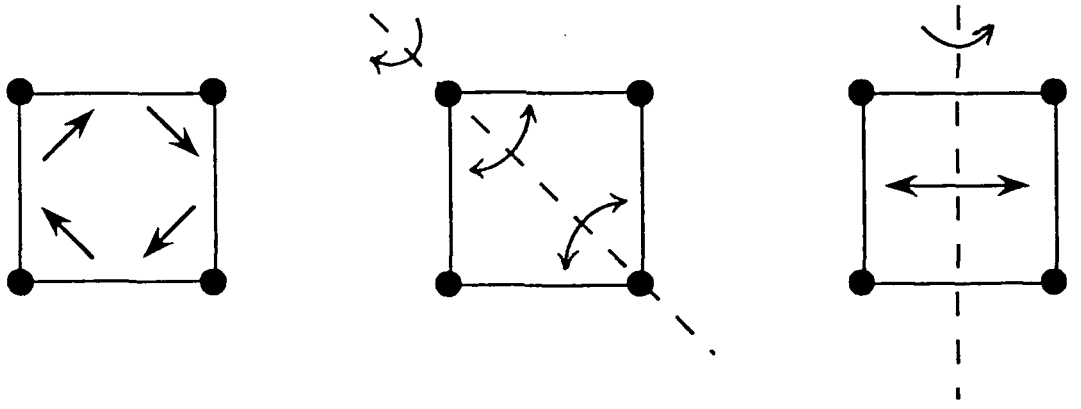


Figure 5.3 Permissible Rigid Motions of a Polygon

2. **Fixed Colorings Under Given Permutations**

Any permutation may be factored into a product of disjoint cycles [Ref 8:p. 69]. In order for a coloring of the polygon to be preserved under that permutation, all edges in a cycle must have the same color. Since opposite edges are to

be colored differently, they must be in disjoint cycles. Then all edges which are opposite the edges of a given cycle must themselves be in a disjoint cycle, which will be called the companion cycle. In order for a coloring to be fixed under a given permutation, each cycle must have a unique companion cycle, hence the number of disjoint cycles must be even. In such cases the edges in one cycle are colored black while those in its companion cycle are colored white. Figure 5.4 provides an example of a coloring which is fixed under a permutation, and shows the companion cycles for that permutation.

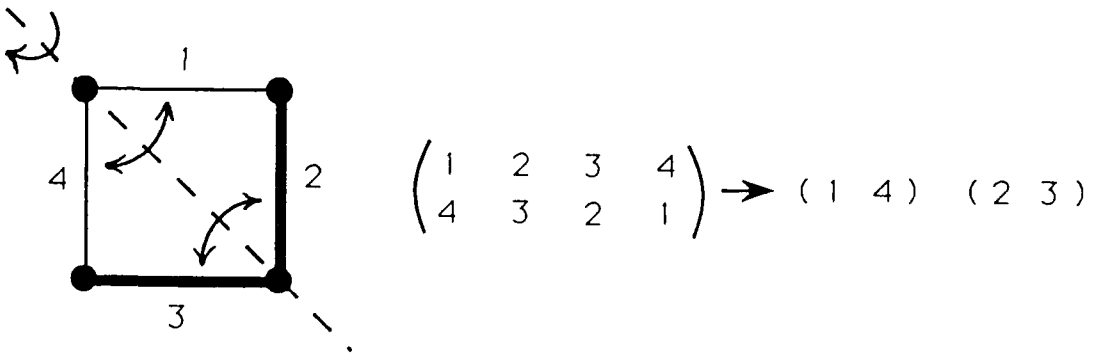


Figure 5.4 Disjoint Cycles Produced by a Permutation

We consider each of the three types of permutations. Throughout the following discussion $\Psi(\pi)$ is defined to be the number of edge two-colorings which remain fixed under a given permutation, π. For simplicity, the edges of each 2n-gon will be numbered clockwise beginning at some fixed edge.

a. Rotations

The identity permutation, π_0, is that motion which does not move the polygon. Hence, $\Psi(\pi_0) = 2^n$. The remaining rotations are through the first 2n-1 multiples of $(360/2n)^\circ$ in the clockwise direction.

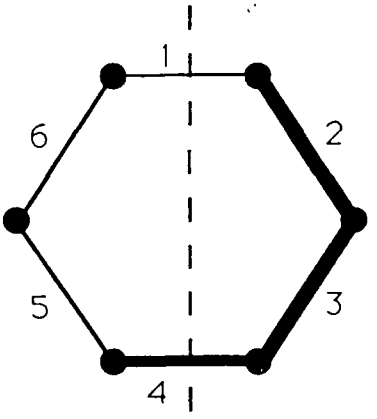
Define a <u>click</u> to be $(1/2n)^{th}$ of a full rotation of the polygon. A rotation, π_i, i = 1,...,(2n-1), may be thought of as turning the polygon i clicks in the clockwise direction, thus moving each edge i positions along from its starting point.

For the disjoint cycles generated by this permutation to have their coloring preserved, the number of disjoint cycles must be even. The edges in half of the cycles may be colored arbitrarily, while fixing the color of those in the companion cycles to the opposite color. Hence the number of choices to be made in coloring the edges of the polygon is one half the number of cycles, and $\Psi(\pi_i) = 2^{(\#cycles/2)}$. If the number of disjoint cycles is odd then $\Psi(\pi_i) = 0$.

b. Reflections About Bisectors

Once a bisector is drawn through two opposite edges of the polygon, the permutation for a reflection about that bisector yields a product of disjoint cycles consisting of two 1-cycles and (n-1) 2-cycles. Such a permutation is illustrated in Figure 5.5. In order for each cycle to have a companion cycle, (n-1) must be even. Hence, n must be odd. In this case coloring half the 2-cycles fixes the color of the

remaining companion cycles, and the number of possible choices for coloring the (n-1) cycles is (n-1)/2. One additional choice may be made in coloring those edges which lie on the bisector, resulting in a total of c = (n+1)/2 choices for coloring. Hence, when n is odd, $\mathbf{Y}(\pi_i) = 2^c$, where i = 2n,...,(3n-1).



(1) (2 6) (3 5) (4)

Figure 5.5 Disjoint Cycles for Reflection About a Bisector

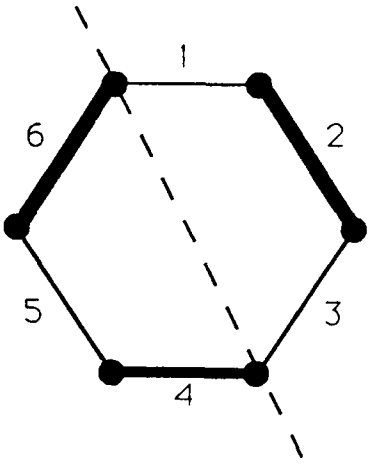
For n even, the number of 2-cycles produced by the permutation is odd. This results in a cycle which fails to have a companion cycle. Hence no coloring may be preserved under the permutation. Thus, when n is even, $\mathbf{Y}(\pi_i) = 0$, i = 2n,...,(3n-1).

c. *Reflections About Diagonals*

The permutation resulting from a reflection about an axis drawn through opposite vertices of the 2n-gon produces a product of disjoint cycles composed of n 2-cycles. Figure 5.6 illustrates such a reflection. It follows

immediately that when n is even, each cycle will have a companion and a total of n/2 choices may be made in coloring the 2n-gon. Then $\Psi(\pi_i) = 2^{n/2}$, i = 3n,...,(4n-1).

On the other hand, when n is odd, there will exist a cycle which does <u>not</u> have a companion cycle, and no coloring may be preserved under the permutation. This implies that when n is odd, $\Psi(\pi_i) = 0$, i = 3n,...,(4n-1).



(1 6) (2 5) (3 4)

Figure 5.6 Disjoint Cycles for Reflection About a Diagonal

3. Burnside's Lemma

At this point, Burnside's Lemma is used to calculate the total number, N, of non-equivalent edge two-colorings of a 2n-gon given the restriction that opposite edges must be colored differently. The lemma is as follows.

> Let E be the set of edges of a regular polygon and let G be a group of permutations, π_i, on the set E. Further, let C be any collection of colorings of E that is closed under G. Then the number, N, of non-equivalent colorings of the edges of the polygon is given by

$$N = \frac{1}{|G|}\sum_{\pi \in G} \Psi(\pi) \quad , \tag{5.1}$$

where $|G|$ is the number of permutations in G and $\Psi(\pi)$ is the number of colorings in C which are left fixed by π.[Ref. 7:p. 343]

Given the information derived earlier, this lemma may be invoked to produce N. This number is given by

$$N = \frac{1}{4n}\sum_{i=0}^{4n-1} \Psi(\pi_i) \quad . \tag{5.2}$$

Hence, the total number of non-isomorphic ring graphs given n fixed obstacles is obtained.

D. PRODUCING NON-ISOMORPHIC RING GRAPHS

Given the number of non-isomorphic ring graphs, N, obtained by applying Burnside's Lemma, the next step is to produce them.

1. Using Partitions of the Integer n

Given n obstacles, we begin with a partition of n into summands, $m_1,\ldots,m_k$, where k is odd. Thus $n = m_1+m_2+\ldots+m_k$. With the summands in some fixed order, an edge two-coloring of the 2n-gon, and therefore a ring graph, may be produced in the following manner. Start at some fixed edge in the 2n-gon and color the first m_1 edges black, proceeding clockwise from the starting edge. Color the next m_2 edges white, the next m_3

edges black, etc., until all k summands have been used. The colors of the remaining n edges become fixed as this procedure is followed.

Using the procedure given above, only partitions with an odd number of summands are used. This guarantees that the starting edge and the $2n^{th}$ edge do not have the same color. Therefore, the two-colorings of the 2n-gon are not equivalent, and the associated ring graphs are not isomorphic. Figure 5.7 illustrates two equivalent two-colorings of an octagon (under rotation through 45 degrees) using two different partitions. The first partition of n uses an odd number of summands, the second partition uses an even number of summands.

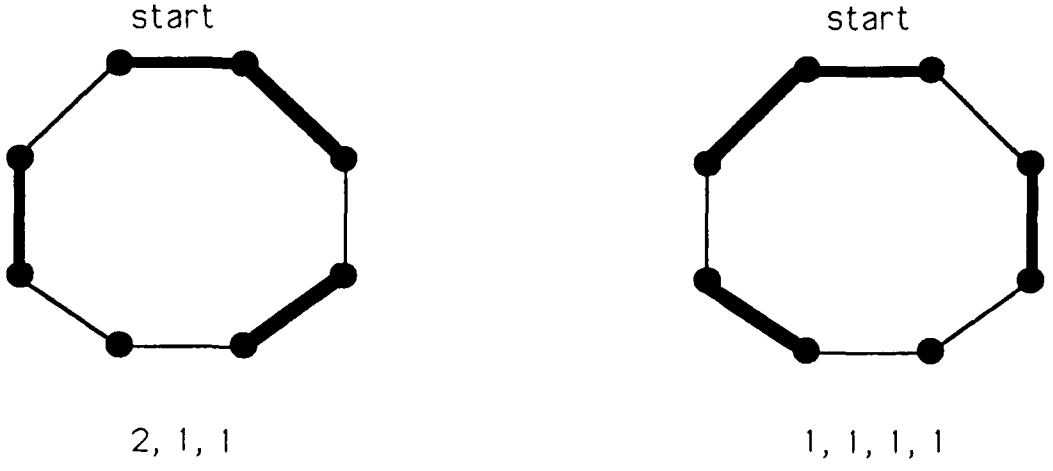


Figure 5.7 Equivalent Two-colorings Resulting From Different Partitions

It is possible that the number of candidate partitions, those with an odd number of summands, may actually

be less than N, the total number of non-isomorphic colorings calculated using Burnside's Lemma. When this occurs, there must exist at least one partition which, when ordered differently, is capable of producing more than one graph which is non-isomorphic to the others. Figure 5.8 shows, for n = 7, two non-equivalent two-colorings constructed from different orderings of the partition whose summands are 2,2,1,1,1. The question which arises is: How many non-equivalent two-colorings, and hence non-isomorphic ring graphs may be produced from a given partition of n into an odd number, k, of summands?

Polya's Pattern Inventory Theorem may be used to answer this question.

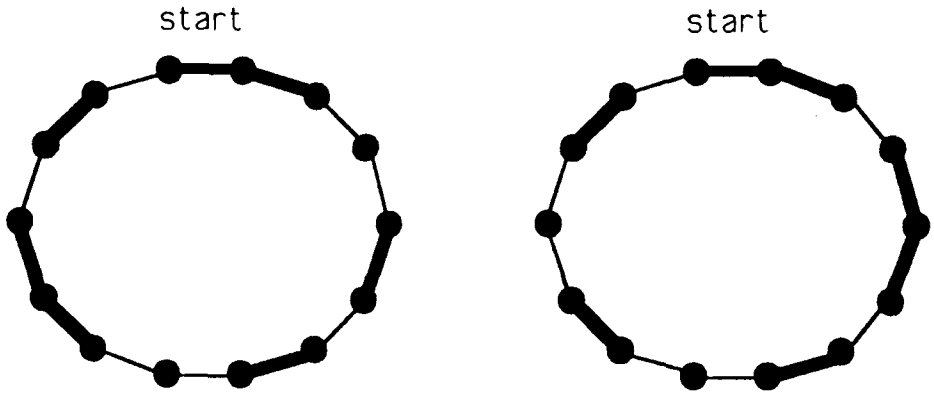


Figure 5.8 Non-Equivalent Colorings Constructed Using the Summands 2,2,1,1,1 of n = 7

2. Polya's Pattern Inventory Theorem

An inventory is a list which indicates how many items of a specific type exist [Ref. 7:p. 333]. This information may be encoded in the coefficients of a polynomial. Polya's Pattern Inventory Theorem follows.

> Let V be the set of vertices of a regular polygon. Let G be a group of permutations on the set V that acts to induce an equivalence relation on the colorings of V. The inventory of non-equivalent colorings of V using two colors is given by the generating function
>
> $$P_G\ ((b+w),\ (b^2+w^2),\ldots,\ (b^k+w^k)),$$
>
> where P_G is the cycle index for the group of symmetries of the polygon. The inventory using colors $c_1, c_2, \ldots, c_n$ is given by
>
> $$P_G\left(\sum_{j=1}^{m} c_j,\ \sum_{j=1}^{m} c_j^2,\ldots,\ \sum_{j=1}^{m} c_j^r\right) \quad . \quad \textit{[Ref. 7:p. 355]} \qquad (5.3)$$

3. Establishing the Cycle Index

We begin with the set of all ordered partitions of n into k summands, where k is an odd integer. Any two partitions will be considered to be equivalent if there exists a cyclic shift of the elements of one partition, a complete reversal of the elements of one partition, or a combination of these which makes the partitions identical. Thus the set of ordered partitions of n may be separated into equivalence classes, with each class producing a non-equivalent two-coloring of the 2n-gon.

The cyclic shifts and reversals of elements in a partition of k summands correspond to the rotations of a k-gon

and reflections of a k-gon about an axis drawn through a vertex and bisecting its opposite edge. Therefore, it is reasonable to consider the number of non-equivalent vertex colorings of a k-gon where the number of colors used is equal to the number of <u>different</u> summands in the partition. For example, with n = 7, a partition of n might be 2,2,1,1,1. Thus, k = 5. Here the vertices of a pentagon would be colored using two colors since only two different summands (1 and 2) appear in the partition.

In examining any k-gon where k is odd, it is seen that k rotations and k reflections are possible for a total of 2k rigid motions. The cycle index for the k-gon given these permutations is then produced.

4. Applying Polya's Theorem

Once the cycle index, P_G, has been established, Polya's Theorem may be applied, where the number of colors, m, is equal to the number of different summands in the given partition. After the cycle index has been expanded using these m colors, the coefficient of the term whose exponents are equal to the multiplicities of each of the integers in the partition of n is determined. This coefficient represents the number of non-isomorphic ring graphs which may be produced from the given partition of n.

The sum of these coefficients, after applying this procedure to <u>all</u> partitions of n with an odd number of summands, is equal to the total number, N, of non-isomorphic

ring graphs given n fixed obstacles. This represents the number of distinct reference frames which may be constructed in the plane with n obstacles, thus generating N different sets of homotopy class names.

E. AN ALTERNATE SOLUTION TO THE COUNTING PROBLEM

Upon completing the research and derivation of the solution method discussed above, an alternate solution was encountered in the literature.

Fredricksen and Kessler [Ref. 9] present a procedure which may be employed in counting and generating the set of non-isomorphic ring graphs. This method is based upon an algorithm which produces the set of all lexicographic compositions of a positive integer, n.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

Determination of the shortest path between two points in the plane containing obstacles has applications in many fields, particularly that of robotic path planning. This thesis presents a computational method which may be employed to generate the names of homotopy classes for a set of paths. By modeling the topological relationships of the region with a graph, a tool is made available for employing this method in producing only the names of those classes which contain candidates for the shortest path.

B. RECOMMENDATIONS FOR FURTHER STUDY

The field of graph theory is one that is gaining vast interest in mathematics and computer science. The graphs which are used to model the regions discussed in this thesis have several interesting characteristics. Further study and analysis of these graphs and their properties may aid future researchers in answering other questions concerning the shortest path problem. For instance, is there a sharp upper bound on the length of the class names which are generated by traversing the graph using the algorithm presented in Chapter III?

Another logical step in the shortest path problem is to extend the analyses presented here to include the third dimension. This extension follows since most industrial robot arms are of a pivoting arm type which are permitted to move freely through space while rotating at some fixed shoulder point.

The computational investigation presented in Appendix A was programmed with knowledge of only elementary FORTRAN programming skills. Although a complexity analysis was not done, it is believed that further study and refinement of the program code could result in a program which is much more efficient.

Finally, this thesis introduces only part of the solution to the shortest path problem. The *thesis* submitted by CPT André M. Cuerington, U.S. Army, presents the remaining portions of the solution, and may be studied in conjunction with this paper.

APPENDIX A. A COMPUTATIONAL INVESTIGATION

A. INTRODUCTION

This appendix describes a computational investigation which was conducted as an aid in verifying that the homotopy class names produced by Algorithm 1, are reasonable when compared to a known method of naming homotopy classes [Ref. 2].

A second algorithm, Algorithm 2, is introduced here to forge a link between the class names of Chapter II and the well-known fundamental group. Two processes are presented to exploit this link: If Algorithm 1 truly names the classes, then both procedures should produce the same output for every path tested. One million test cases were examined and no counter-examples were found. While this computational evidence is not a proof, it does support the proof of the claim that Algorithm 1 names homotopy classes [Ref. 2].

B. ALGORITHM 2

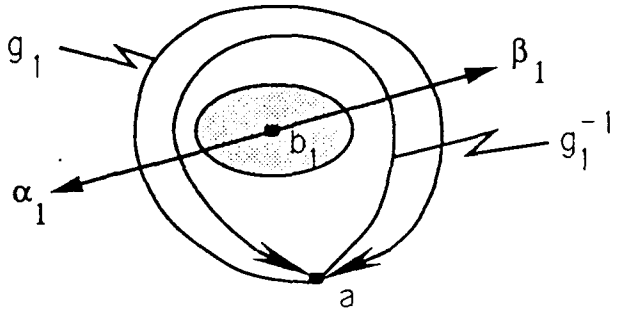
1. Fundamental Group

The class name obtained from Algorithm 1 was written in terms of the alphabet A defined in Chapter II. We present an algorithm here where the name obtained is expressed in terms of the fundamental group of a topological space T.

The basic idea for producing a fundamental group for this problem is to regard the paths in T as elements of the group, with path concatenation (*) as the group operation. There are two problems with considering the paths as the group elements. First, it is not necessarily possible to concatenate any two paths in T. In order to concatenate two paths using the operation (*), the first path must end at the point where the second begins. To make it possible to concatenate paths, in this analysis the test paths all begin and end at the point a. This point is called a base point.

The second problem to overcome is that an inverse is not well defined for paths. If the identity is defined as the empty string, i.e., stay at the base point, then when a path and its inverse are concatenated we have traced out a path which is not equal to the identity. To avoid this problem, the homotopy classes of paths are considered. Then all paths that have an empty character string representation will be in the identity class. Also, any class followed by its inverse will equal the identity. In Figure A.1, g_1 represents the class $\alpha_1\beta_1$, and $(g_1)^{-1}$ represents the class $\beta_1\alpha_1$. So, $g_1 * (g_1)^{-1}$ represents the class $\alpha_1\beta_1\beta_1\alpha_1$ which reduces to the identity class by the rules of Algorithm 1.

The group elements, therefore, are homotopy classes of loops around obstacles based at a common point, and the group operation is defined in terms of concatenation (*) of these classes.



g_1 : class $\alpha_1\beta_1$

g_1^{-1} : class $\beta_1\alpha_1$

$g_1 * g_1^{-1}$: $\beta_1\alpha_1\alpha_1\beta_1 = e$

Figure A.1 A Homotopy Class and its Inverse

Given a base point a, let p be any loop beginning and ending at a. Let $[p_i]$ denote the homotopy class of p_i. So the set of group elements is {$[p_i]$ such that p_i is a member of T, where p_i is a loop based at a}. This set will be called G with the elements denoted g_j. Figure A.2 illustrates several paths and their homotopy classes given n = 2 obstacles.

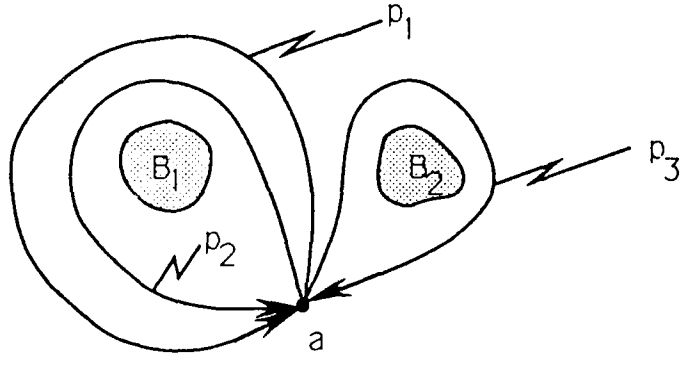


Figure A.2 $[p_1] = [p_2] * [p_3]$

Now the group operation (*) can be defined by

$[p] * [q] = [pq]$.

It is important to note that the fundamental group is finitely generated. As Figure A.3 illustrates, the generators are not unique. By fixing a set of generators, however, the class names become fixed.

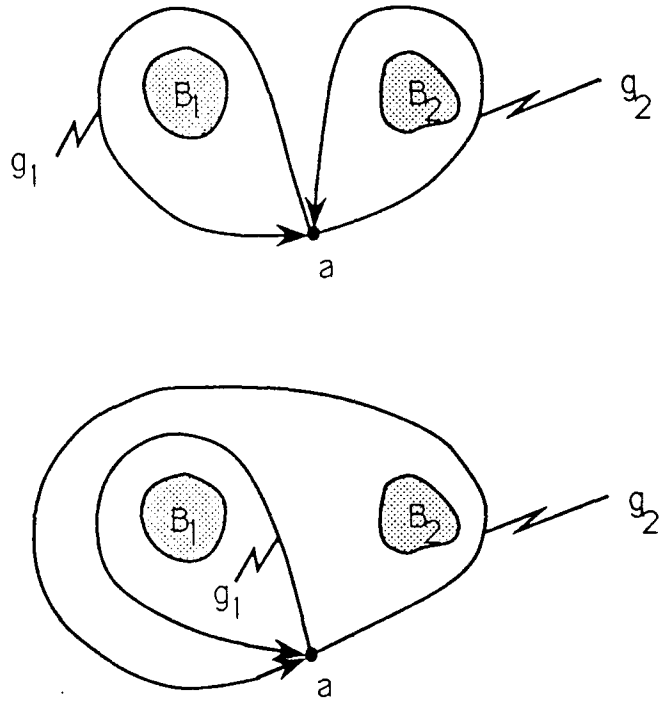


Figure A.3 Alternate Generators For Fundamental Group of the Space With Two Obstacles

With the above information describing the fundamental group representation, Algorithm 2 is given below.

2. Functions in Algorithm 2

a. *Side Array*

Consider the reference lines L_k constructed in Chapter II to be oriented rays in the direction from c to b_k. It then becomes reasonable to discuss a 'right' and a 'left' side of those lines. We allow a moving point to trace a path p from start point to destination point. Side(k) is a function which defines the side of L_k on which the moving point lies. The output is either 'left' or 'right'. The output is never 'on' because this routine is used only after a complete crossing of L_k takes place. A crossing is considered to be complete when the moving point leaves L_k to one side after having met L_k from the opposite side.

b. *Switch Function*

The switch function is defined by:

$$\text{switch (side(k))} = \begin{cases} \text{'right' if side (k) = 'left'} \\ \text{'left' if side (k) = 'right'} \end{cases}$$

As the moving point traces p, each time a reference line L_k is crossed, the switch function will be applied to indicate on which side of L_k the moving point lies.

c. *Index Function*

Let x_k be any character from the alphabet A representing the k^{th} element in R(p). Let index(x) = j if x equals α_j or β_j; thus, the k^{th} crossing completed by the moving point is a crossing of L_j. So if the moving point is on the

left side of L_3 initially and this line is now crossed on either side of b_3, then side(3) would equal 'right'.

d. The Algorithm

Using the functions described above, Algorithm 2 is given in Figure A.4. The algorithm works in two phases. The first phase initializes the array side(k) for k = 1,...,n. The second phase reads the raw string from left to right and adds an element to the fundamental group representation for each ß crossing. When the end of the input string is reached the output, F(R(p)), is a shorter string with one character corresponding to every ß element in the original raw string. Each character represents a generator or its inverse.

```
begin
input R(p) = x1x2...xm
for j = 1,..., n
    If a is to right of reference line Lj then
        side (j) ← right
    else
        side (j) ← left
    end if
end for
G ← [a]
for k = 1,..., m
    i = index(xk)
    side(i) ← switch(side(i))
    if xk = ßr for some r, then
        if side(index(xk)) = left, then
            G ← G * gi
        else
            G ← G * (gi)^-1
        end if
    end if
end for
F(R(p)) ← G
end
```

Figure A.4 Algorithm 2

e. *Fundamental Group Cancellation Function*

The cancellation rules in the fundamental group differ somewhat from those for the raw string. Although sets of generators are not unique, we can obtain a unique representation of each class with respect to a particular set of generators. For any given set of generators, every homotopy class can be represented as a product of these generators and their inverses. Even so, this representation is not unique until cancellation is applied. The cancellation rule follows.

Let $G = \{g_1, \ldots, g_n\}$ be a set of generators of the fundamental group of T (base point a) and let $Y = y_1 y_2 \ldots y_m$ be a representation of some homotopy class in terms of the g_j and their inverses, i.e. for each $i = 1, \ldots, m$, $y_i = g_j$ or $(g_k)^{-1}$ for some $j,k = 1, \ldots, n$.

The cancellation function κ is defined as follows. If Y contains a two character substring $y_i y_{i+1}$ with $y_i = (y_{i+1})^{-1}$ then $\kappa(Y)$ is the string which results by removing both y_i and Y_{i+1}, in their leftmost occurrence, from Y. Finitely many repeated applications of κ produce a string in which no further cancellation is possible. We define this string as $K(Y)$.

C. THE COMPUTATIONAL INVESTIGATION

1. The Approach

A graphical representation of the two procedures is provided in Figure A.5. The investigation begins by accepting a random path p as input and generating two names for the homotopy class representing p which are output.

The idea of the test is to determine for a given path its raw string character representation. For computational purposes, the paths considered are polygonal paths. This raw string is then input to both algorithms.

The raw string (R(p)) is first entered into Algorithm 2. The output (F(R(p))) from Algorithm 2 is then subject to the cancellation routine which outputs the fully cancelled string (K(F(R(p)))). Next, R(p) is input to Algorithm 1 which produces the canonical string (C(R(p))). Algorithm 2 is then applied to C(R(p)) to output F(C(R(p))). The results of these tests are then compared, and the program continues to run until a predetermined number of paths are checked.

It can be shown that C(R(p)) is a unique class name if and only if F(C(R(p))) equals K(F(R(p))) for all p in T [Ref. 2]. Before the proof of this conjecture was obtained, the above procedures were programmed and tested for one million different paths and no counter-examples were discovered.

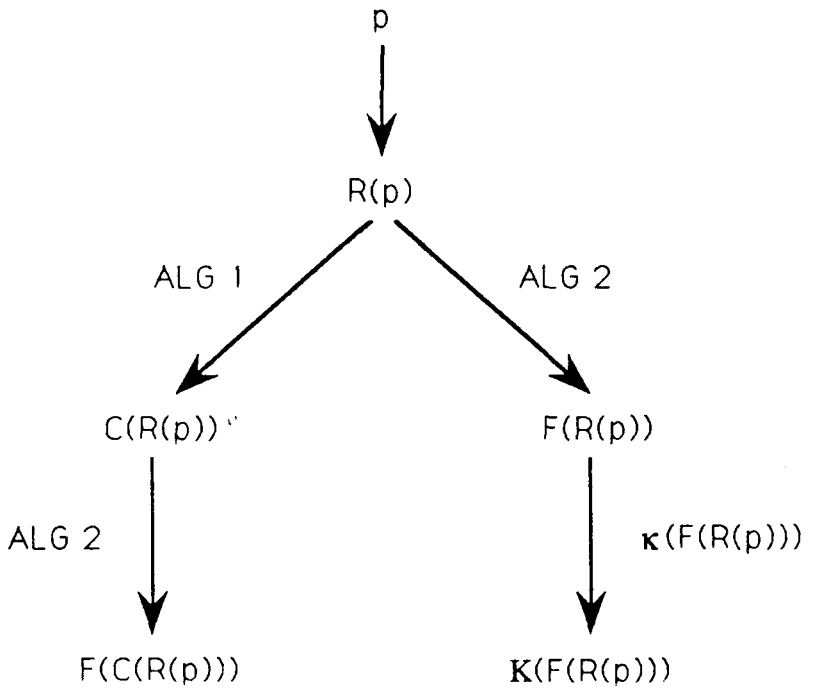


Figure A.5 Flow Chart of Two Algorithms

2. The Test

The general algorithm used to test the two procedures represented in Figure A.5 is easily followed and is shown in Figure A.6. However, the attached program, which was used to test the model, is substantially more involved. This was done in an attempt to write efficient programming code.

Two time saving techniques are employed in order to test the one million cases in 25 minutes. The first method involves subroutines which perform cancellation. It is simple to code a program that scans the character representation repeatedly to find all possible cancellation in the string $\beta_1\beta_2\alpha_1\alpha_1\beta_2$, which reduces to β_1. However, in the enclosed code,

pointers are used to mark the position where the first cancellation occurs and then check the newly adjacent characters for cancellation. This method reduces the complexity of an algorithm and is applied in both Algorithm 1 and Algorithm 2.

```
n = number of reference frames to be considered
m = number of paths to be considered
nobs = number of obstacles on each board
seed = input seed for random generator (not needed?)

begin
for board = 1,..., n
    create reference frame or board
    for path = 1,..., n
        1. Create a polygonal path p with nseg
           segments
        2. Form R(p)
        3. Form F(R(p))
        4. Form K(F(R(p)))
        5. Form C(R(p))
        6. Form F(C(R(p)))
        if K(F(R(p))) does not equal F(C(R(p))) then
            print (raw string, board, and path info
          for the counter example)
        end if
    end for
end for
print (final seed)
end
```

Figure A.6 Test Algorithm

The second time-saving device involves the sorting of character strings. Instead of a bubble sort method (order n^2 complexity), a merge sort algorithm (order n(log(n)) complexity) is used. As can be seen in the attached code, this choice to save computer time calls for a significant

increase in programmer time, which is generally much more expensive.

The code which is used to implement this test is presented in the remaining pages of this appendix.

```
      PROGRAM CLASNAME FORTRAN

C ***************************************************************************
C  THIS PROGRAM RANDOMLY GENERATES OBSTACLES AND A POLYGONAL PATH
C  THROUGH THOSE OBSTACLES TO DETERMINE WHETHER OR NOT THE PROCEDURES
C  USED BY COMPETING ALGORITHMS PRODUCE THE SAME FUNDAMENTAL GROUP
C  REPRESENTATION FOR THAT PATH.
C ***************************************************************************

C  THIS PORTION OF THE PROGRAM SERVES AS THE MAIN DRIVER WHICH RECEIVES
C  THE PARAMETERS DEFINING THE REGION AND INITIALIZES THE LINK LIST
C  ARRAYS WHICH WILL BE USED TO REPRESENT THE POLYGONAL PATH.

C  INPUT:  INITIAL SEED FOR THE RANDOM NUMBER GENERATOR, NUMBER OF
C          OBSTACLE CONFIGURATIONS AND PATHS TO BE TESTED, NUMBER OF
C          OBSTACLES IN THE PLANE AND NUMBER OF SEGMENTS IN EACH
C          POLYGONAL PATH

C  OUTPUT:  FINAL RANDOM NUMBER GENERATOR SEED AND MESSAGES INDICATING
C           ANY PATHS WHICH PRODUCE DIFFERENT FUNDAMENTAL GROUP
C           REPRESENTATIONS

      REAL*8   BX(1000), BY(1000), X(1000), Y(1000), DSEED
      INTEGER  NOBS, NSEGS, NUMPTS, N, M, HEAD(1000)
     +         NEXT(1000), PRED(1000), BOARD, PATH
      EXTERNAL GGUBFS

      DSEED = 123457.0
      M = 1000
      N = 1000
      NOBS   = 20
      NSEGS  = 5
      NUMPTS = NSEGS + 1
      PRINT*,'INPUT SEED', DSEED

      DO 1 BOARD = 1, N

          CALL BOARDS (DSEED, BX, BY, NOBS)
          DO 2 PATH = 1, M

             CALL INIT (HEAD, NEXT, PRED, PSEED, DSEED )
             CALL PATHS (DSEED, X, Y, NSEGS)
             CALL TEST(HEAD,NEXT,PRED,BX,BY,NOBS,X,Y,NUMPTS,BSEED,PSEED)

    2     CONTINUE
    1 CONTINUE

      PRINT*,' '
      PRINT*,'FINAL SEED', DSEED

      STOP
      END
```

```
      SUBROUTINE INIT ( HEAD, NEXT, PRED , PSEED, DSEED)
C ***************************************************************************

C  THIS SUBROUTINE INITIALIZES THE DOUBLE LINK LIST ARRAYS THAT WILL
C  REPRESENT THE PATH.

C  INPUT:  DSEED

C  OUTPUT: HEAD, NEXT, AND PRED ARRAYS SET TO ZERO, AND SEED FOR THE
C  R.N.G. PRIOR TO CONSTRUCTION OF THE PATH

      INTEGER HEAD(1000), NEXT(1000), PRED(1000)
      REAL*8 PSEED, DSEED

      PSEED = DSEED

      DO 3 K = 1,1000
           HEAD(K) = 0
           NEXT(K) = 0
           PRED(K) = 0
    3 CONTINUE

      RETURN
      END



      SUBROUTINE BOARDS (DSEED, BX, BY, NOBS)
C ***************************************************************************

C  THIS SUBROUTINE USES A PSEUDO RANDOM NUMBER GENERATOR TO CREATE THE
C  COORDINATES OF EACH OBSTACLE ON THE BOARD, SCALING ALL COORDINATES
C  TO BE IN THE INTERVAL (-1,1).

C  INPUT:  NUMBER OF OBSTACLES IN THE REGION AND A SEED FOR THE R.N.G.

C  OUTPUT:  OBSTACLE COORDINATES

      REAL*8 BX(1000), BY(1000), DSEED
      EXTERNAL GGUBFS

      DO 1 I = 1, NOBS
          BX(I) = 2. * GGUBFS(DSEED) - 1.
    1     BY(I) = 2. * GGUBFS(DSEED) - 1.

      RETURN
      END
```

```
      SUBROUTINE PATHS (DSEED, X, Y, NSEGS)
C ***************************************************************************

C  RANDOMLY GENERATES THE X AND Y COORDINATES OF THE VERTICES FOR THE
C  POLYGONAL PATH, SCALING ALL COORDINATES TO BE IN THE INTERVAL (-1,1).
C  THIS SUBROUTINE ALSO ENSURES THAT THE PATH IS A CLOSED LOOP BY
C  ASSIGNING THE START/FINISH POINTS THE SAME COORDINATES.

C  INPUT:  NUMBER OF PATH SEGMENTS AND A SEED FOR THE R.N.G.

C  OUTPUT:  COORDINATES OF VERTICES ALONG THE POLYGONAL PATH

      REAL*8 X(1000), Y(1000), DSEED
      EXTERNAL GGUBFS

      DO 1 I = 1, NSEGS
          X(I) = 2. * GGUBFS(DSEED) - 1.
    1     Y(I) = 2. * GGUBFS(DSEED) - 1.

      X(NSEGS+1) = X(1)
      Y(NSEGS+1) = Y(1)

      RETURN
      END



      SUBROUTINE TEST(HEAD, NEXT, PRED, BX, BY, NOBS, X, Y, NUMPTS,
     +                BSEED, PSEED)
C ***************************************************************************

C  GENERATES THE RAW STRING OF CHARACTERS REPRESENTING A PATH AND
C  DETERMINES ITS FUNDAMENTAL GROUP REPRESENTATION USING ALGORITHMS 1
C  AND 2.  RESULTS OF THESE COMPETING ALGORITHMS ARE THEN COMPARED FOR
C  DIFFERENCES.

C  INPUT:  A POLYGONAL PATH AND COORDINATES OF POINTS REPRESENTING
C          OBSTACLES

C  OUTPUT:  COMPARED RESULTS OF ALGORITHMS

      INTEGER HEAD(1000), NEXT(1000), PRED(1000), FR(1000), NR(1000),
     +        PR(1000), FL(1000), NL(1000), PL(1000)
      REAL*8  BX(1000), BY(1000), X(1000), Y(1000)

      DATA FR/1000*0/
      DATA NR/1000*0/
      DATA PR/1000*0/
      DATA FL/1000*0/
      DATA NL/1000*0/
      DATA PL/1000*0/
```

```
      CALL RAWSTR(BX, BY, NOBS, X, Y, NUMPTS, HEAD, NEXT, PRED, NELEMS)
      CALL ALG2 (NOBS, NELEMS, HEAD, NEXT, X, Y, BX, BY, FR, NR, PR)
      CALL CANALG2 (FR, NR, PR)
      CALL ALG1 (HEAD, NEXT, PRED, NELEMS)
      CALL ALG2 (NOBS, NELEMS, HEAD, NEXT, X, Y, BX, BY, FL, NL, PL)
      CALL CTREX (FL, NL, PL, FR, NR, PR, BSEED, PSEED)

      RETURN
      END



      SUBROUTINE RAWSTR(BX, BY, NOBS, X, Y, NUMPTS, HEAD, NEXT, PRED,
     +                  NELEMS)
C ***************************************************************************

C  PRODUCES THE RAW STRING OF CHARACTERS WHICH REPRESENTS THE PATH,
C  PAYING CLOSE ATTENTION TO THE ORDERING OF THE CHARACTERS WHERE
C  NECESSARY.  THE STRING IS CONSTRUCTED BY IDENTIFYING THOSE OBSTACLE
C  REFERENCE LINES WHICH ARE CROSSED AS EACH PATH SEGMENT IS TRAVERSED
C  IN ORDER.

C  INPUT:  ORDERED LIST OF VERTICES REPRESENTING THE PATH AND A POINT
C          REPRESENTING EACH OBSTACLE IN THE REGION

C  OUTPUT:  A RAW STRING OF ALPHAS AND BETAS CONTAINED IN AN ARRAY
C           NAMED HEAD() AND ITS PARALLEL ARRAYS NEXT() AND PRED() WHICH
C           PRODUCE THE DOUBLE LINKED LIST

      IMPLICIT REAL*8 (A-H,O-Z)
      INTEGER  HEAD, NEXT, SEG, FSTSTR, HDINDX, SEGEND, FIRST, START,
     +         LENGTH, PRED
      LOGICAL ALLALF
      DIMENSION X(1000), Y(1000), BX(1000), BY(1000), A1(1000),
     +          B1(1000), A2(1000), B2(1000), D2(1000), DIST(1000),
     +          FIRST(1000), HEAD(1000), NEXT(1000), PRED(1000)

      DATA DIST/1000*0.0/
      DATA FIRST/1000*0/

      CALL SETUP(NUMPTS,NOBS,CX,CY,X,Y,BX,BY,A1,B1,A2,B2,D2)

      HDINDX = 2
      NSEGS = NUMPTS - 1
      LOLD = 1
      HEAD(LOLD) = 0
```

```
      DO 7 SEG = 1, NSEGS
          SEGEND = SEG + 1
          FSTSTR = HDINDX
          LSTSTR = HDINDX
          ALLALF = .TRUE.
          LENGTH = 0

          DO 6 LINE = 1, NOBS
              CHECK1 = (A1(LINE)*X(SEG)+B1(LINE)*Y(SEG))*
     +                 (A1(LINE)*X(SEGEND)+B1(LINE)*Y(SEGEND))
              IF (CHECK1.LT.0) THEN
                  LENGTH = LENGTH + 1
                  CHECK2 = (A2(SEG)*BX(LINE)+B2(SEG)*
     +                     BY(LINE)+D2(SEG))*D2(SEG)
                  IF (CHECK2.LT.0) THEN
                      HEAD(HDINDX) = -LINE
                      LSTSTR = HDINDX
                      HDINDX = HDINDX + 1
                  ELSE
                      CALL CASES1 (A1, B1, A2, B2, D2, SEG, LINE, XINT,
     +                             YINT)
                      DISTC = XINT**2 + YINT**2
                      DISTB = (XINT-BX(LINE))**2 + (YINT-BY(LINE))**2
                      IF (DISTB.LT.DISTC) THEN
                          HEAD(HDINDX) = LINE
                          LSTSTR = HDINDX
                          HDINDX = HDINDX + 1
                          ALLALF = .FALSE.
                      ELSE
                          HEAD(HDINDX) = -LINE
                          LSTSTR = HDINDX
                          HDINDX = HDINDX + 1
                      ENDIF
                  ENDIF
              ENDIF
    6     CONTINUE
          IF (LENGTH.NE.0) THEN
              IF (ALLALF) THEN
                  CALL ALPHAS (NEXT, LOLD, FSTSTR, LSTSTR, SEG, START,
     +                         HEAD, NSEGS, PRED)
              ELSE
                  CALL ORDER(SEG, LINE, HDINDX, HEAD, NEXT, A1, A2,
     +                       B1, B2, D2, FSTSTR, LSTSTR, X, Y, START,
     +                       LENGTH, FIRST, LOLD, NSEGS, PRED)
              ENDIF
          ENDIF
    7 CONTINUE

      CALL COUNTR(START, NEXT, HEAD, NELEMS)
      RETURN
      END
```

```
      SUBROUTINE SETUP(NUMPTS, NOBS, CX, CY, X, Y, BX, BY, A1, B1, A2,
     +                 B2, D2)
C ***************************************************************************

C  FOR EACH OBSTACLE, THIS ROUTINE DETERMINES THE COEFFICIENTS OF THE
C  EQUATION FOR THE REFERENCE LINE FROM THE OBSTACLE TO THE ORIGIN.  IN
C  ADDITION, IT CALCULATES THE COEFFICIENTS OF THE LINE REPRESENTING
C  EACH SEGMENT OF THE POLYGONAL PATH.

C  INPUT:  NUMPTS, NOBS, COORDINATES OF VERTICES ALONG POLYGONAL PATH
C          AND COORDINATES OF THOSE POINTS WHICH REPRESENT EACH
C          OBSTACLE

C  OUTPUT:  COEFFICIENTS OF LINEAR EQUATIONS REPRESENTING PATH SEGMENTS
C           AND REFERENCE LINES FOR EACH OBSTACLE

      IMPLICIT REAL*8 (A-H, O-Z)
      DIMENSION X(1000), Y(1000), BX(1000), BY(1000), A1(1000),
     +          B1(1000), A2(1000), B2(1000), D2(1000)

      DO 4 I = 1, NOBS
          A1(I) = BY(I)
          B1(I) = -BX(I)
    4 CONTINUE

      CX = 0.0
      CY = 0.0
      I = 1

      DO 5 J = 2,NUMPTS
          A2(I) = Y(J) - Y(I)
          B2(I) = X(I) - X(J)
          D2(I) = (Y(I)*X(J))-(X(I)*Y(J))
          I = I+1
    5 CONTINUE

      RETURN
      END
```

```
      SUBROUTINE CASES1(A1, B1, A2, B2, D2, SEG, LINE, XINT, YINT)
C ***************************************************************************

C  THIS SUBROUTINE DETERMINES THE COORDINATES FOR THE POINT OF
C  INTERSECTION OF A GIVEN PATH SEGMENT AND A GIVEN OBSTACLE REFERENCE
C  LINE.  NUMERICAL STABILITY OF CALCULATIONS REQUIRES THE MATHEMATICAL
C  OPERATIONS TO BE SEPARATED INTO CASES, THE BEST CASE BEING USED FOR
C  EACH PARTICULAR SITUATION.

C  INPUT:  COEFFICIENTS OF LINEAR EQUATIONS FOR PATH SEGMENT AND
C          REFERENCE LINE TO BE EVALUATED

C  OUTPUT:  POINT OF INTERSECTION OF THE PATH SEGMENT AND THE OBSTACLE
C           REFERENCE LINE

      IMPLICIT REAL*8 (A-H,O-Z)
      INTEGER SEG
      DIMENSION A1(1000), B1(1000), A2(1000), B2(1000), D2(1000), A(2,2)

      A(1,1) = A1(LINE)
      A(1,2) = B1(LINE)
      A(2,1) = A2(SEG)
      A(2,2) = B2(SEG)
      BIGEST = 0.0

      DO 1 L = 1,2
          DO 1 K = 1,2
              TEST = DABS(A(K,L))
              IF (TEST.GT.BIGEST) THEN
                  BIGEST = TEST
                  KBIG = K
                  LBIG = L
              ENDIF
    1 CONTINUE

      IF (KBIG.EQ.1) THEN
          IF (LBIG.EQ.1) THEN
              YINT = -D2(SEG)/(B2(SEG)-B1(LINE)*A2(SEG)/A1(LINE))
              XINT = -B1(LINE)*YINT/A1(LINE)
              RETURN
          ELSE
              XINT = -D2(SEG)/(A2(SEG)-A1(LINE)*B2(SEG)/B1(LINE))
              YINT = -A1(LINE)*XINT/B1(LINE)
              RETURN
          ENDIF
      ELSE
          IF (LBIG.EQ.1) THEN
              YINT = (D2(SEG)*A1(LINE)/A2(SEG))/
     +                (B1(LINE)-B2(SEG)*A1(LINE)/A2(SEG))
              XINT = (-D2(SEG)-B2(SEG)*YINT)/A2(SEG)
              RETURN
```

```
          ELSE
              XINT = (D2(SEG)*B1(LINE)/B2(SEG))/
     +                (A1(LINE)-A2(SEG)*B1(LINE)/B2(SEG))
              YINT = (-D2(SEG)-XINT*A2(SEG))/B2(SEG)
              RETURN
          ENDIF
       ENDIF

       END


       SUBROUTINE ALPHAS (NEXT, LOLD, FSTSTR, LSTSTR, SEG, START, HEAD,
      +                   NSEGS, PRED)
C ***************************************************************************

C  GIVEN THAT A SEGMENT OF THE PATH CROSSES ONLY ALPHA RAYS WHEN
C  TRAVERSED, THESE CROSSINGS ARE SIMPLY INSERTED INTO THE LINK LIST
C  ARRAYS IN THE ORDER IN WHICH THEY WERE DETECTED, I.E., SMALLEST TO
C  LARGEST IN ABSOLUTE VALUE.

C  INPUT:  STRING REPRESENTING THE REFERENCE LINES CROSSED BY THE
C          CURRENT PATH SEGMENT, GIVEN THAT ALL CROSSINGS ARE ALPHAS

C  OUTPUT:  UPDATED NEXT() AND PRED() ARRAYS WHICH CONTAIN THE STRING
C           REPRESENTING THE MOST CURRENT PATH SEGMENT

       INTEGER NEXT(1000), FSTSTR, SEG, START, HEAD(1000), PRED(1000)

       NEXT(LOLD) = FSTSTR
       PRED(FSTSTR) = LOLD
       LAST = LSTSTR - 1

       IF (LAST.GT.FSTSTR) THEN
           DO 12 I = FSTSTR,LAST
               NEXT(I) = I + 1
               PRED(I+1) = I
   12      CONTINUE
       ELSEIF (LAST.EQ.FSTSTR) THEN
           NEXT(FSTSTR) = LSTSTR
           PRED(LSTSTR) = FSTSTR
       ENDIF

       IF(SEG.EQ.1) START = LOLD

       LOLD = LSTSTR
       RETURN
       END
```

```
      SUBROUTINE ORDER(SEG, LINE, HDINDX, HEAD, NEXT, A1, A2, B1, B2,
     +                 D2, FSTSTR, LSTSTR, X, Y, START, LENGTH, FIRST,
     +                 LOLD, NSEGS, PRED)
C ***************************************************************************

C  GIVEN THAT A SEGMENT OF THE PATH CROSSES ONE OR MORE BETAS, THIS
C  SUBROUTINE DETERMINES THE ACTUAL ORDER OF CROSSING WHEN TRAVERSING
C  THE SEGMENT FROM BEGINNING TO END.  THIS IS DONE BY FIRST DETERMINING
C  THE DISTANCE FROM THE SEGMENT START POINT TO THE POINT OF
C  INTERSECTION OF EACH CROSSED OBSTACLE REFERENCE LINE.  THESE
C  DISTANCES ARE THEN SORTED FROM SMALLEST TO LARGEST AND CROSSINGS
C  ARE UPDATED IN THE LINK LIST ACCORDINGLY.

C  INPUT:  HEAD() AND ALL COORDINATES REQUIRED TO DETERMINE THE
C          DISTANCES FROM INITIAL VERTEX OF PATH SEGMENT TO EACH OF
C          THE CROSSED REFERENCE LINES

C  OUTPUT:  HEAD(), NEXT() AND PRED() ARRAYS CONTAINING THE RAW STRING
C           WHICH ACCURATELY REPRESENTS THE PATH ALONG THE CURRENT
C           SEGMENT

      INTEGER  SEG, LINE, HEAD(1000), NEXT(1000), FSTSTR, HDINDX,
     +         START, FIRST(1000), PRED(1000), F
      REAL*8 XINTER(1000), YINTER(1000), A1(1000), B1(1000),
     +       A2(1000), B2(1000), D2(1000), DIST(1000), X(1000), Y(1000)

      DO 8 J = FSTSTR, LSTSTR
          LINE = ABS(HEAD(J))
          CALL CASES2 (A1, B1, A2, B2, D2, SEG, LINE, XINTER, YINTER)
          DIST(J) = (XINTER(LINE)-X(SEG))**2+(YINTER(LINE)-Y(SEG))**2
          DIST(J) = -DIST(J)
          NEXT(J) = J+1
          PRED(J+1) = J
    8 CONTINUE

      NEXT(LSTSTR) = 0
      PRED(LSTSTR+1) = LSTSTR

      CALL MERG2(DIST, NEXT, PRED, FSTSTR, F)

      NEXT(LOLD) = F
      PRED(F) = LOLD
      IF(SEG.EQ.1) START = LOLD
      LOLD = NEXT(LOLD)
   17 IF(NEXT(LOLD).NE.0) THEN
          LOLD = NEXT(LOLD)
          GO TO 17
      ENDIF

      RETURN
      END
```

```
      SUBROUTINE CASES2(A1, B1, A2, B2, D2, SEG, LINE, XINTER, YINTER)
C ***************************************************************

C  DETERMINES COORDINATES FOR THE POINT OF INTERSECTION OF THE PATH
C  SEGMENT AND EACH OF THE REFERENCE LINES IT CROSSES.  AGAIN, IN ORDER
C  TO MAINTAIN NUMERICAL STABILITY, CALCULATIONS ARE MADE USING THE
C  MOST APPROPRIATE CLOSED FORM EQUATION.

C  INPUT:  COEFFICIENTS OF LINEAR EQUATIONS FOR CURRENT PATH SEGMENT
C          AND ALL OBSTACLE REFERENCE LINES WHICH IT CROSSES

C  OUTPUT:  COORDINATES FOR POINTS OF INTERSECTION OF EACH OBSTACLE
C           REFERENCE LINE WITH THE PATH SEGMENT

       IMPLICIT REAL*8 (A-H, O-Z)
       INTEGER SEG
       DIMENSION A1(1000), B1(1000), A2(1000), B2(1000), D2(1000),
      +          XINTER(1000), YINTER(1000), A(2,2)

       A(1,1) = A1(LINE)
       A(1,2) = B1(LINE)
       A(2,1) = A2(SEG)
       A(2,2) = B2(SEG)
       BIGEST = 0.0

       DO 1 L = 1,2
           DO 1 K = 1,2
               TEST = DABS(A(K,L))
               IF (TEST.GT.BIGEST) THEN
                   BIGEST = TEST
                   KBIG = K
                   LBIG = L
               ENDIF
     1 CONTINUE

       IF (KBIG.EQ.1) THEN
           IF (LBIG.EQ.1) THEN
               YINTER(LINE)= -D2(SEG)/(B2(SEG)-B1(LINE)*A2(SEG)/A1(LINE))
               XINTER(LINE)= -B1(LINE)*YINTER(LINE)/A1(LINE)
               RETURN
           ELSE
               XINTER(LINE)= -D2(SEG)/(A2(SEG)-A1(LINE)*B2(SEG)/B1(LINE))
               YINTER(LINE)= -A1(LINE)*XINTER(LINE)/B1(LINE)
               RETURN
           ENDIF
       ELSE
           IF (LBIG.EQ.1) THEN
               YINTER(LINE) = (D2(SEG)*A1(LINE)/A2(SEG))/
      +                       (B1(LINE)-B2(SEG)*A1(LINE)/A2(SEG))
```

```
            XINTER(LINE) = (-D2(SEG)-B2(SEG)*YINTER(LINE))/A2(SEG)
             RETURN
          ELSE
             XINTER(LINE) = (D2(SEG)*B1(LINE)/B2(SEG))/
     +                    (A1(LINE)-A2(SEG)*B1(LINE)/B2(SEG))
             YINTER(LINE) = (-D2(SEG)-XINTER(LINE)*A2(SEG))/B2(SEG)
             RETURN
          ENDIF
         ENDIF
         END



      SUBROUTINE MERG2 (DIST, NEXT, PRED, FSTSTR, F)
C ***************************************************************************

C  THIS SUBROUTINE SORTS A SUBSTRING OF ALL POSTIVE INTEGERS INTO
C  INCREASING ORDER.

C  INPUT:  A DOUBLE LINK LIST CONSISTING OF DIST, NEXT, AND PRED ARRAYS

C  OUTPUT: A DOUBLE LINK LIST WITH ALL ENTRIES PLACED IN THE ORDER
C          IN WHICH THEIR RESPECTIVE REFERENCE LINES WERE CROSSED

       IMPLICIT INTEGER(A-Z)
       REAL*8 DIST(1000)
       LOGICAL DONE
       DIMENSION  NEXT(1000), PRED(1000)

       DONE = .FALSE.
       P = FSTSTR

     1 F = P

       CALL SORT2 (F, PREDF, P, DIST, NEXT, DONE, PRED)

       IF(DONE) RETURN

       GOTO 1

       END
```

```
      SUBROUTINE SORT2(F, PREDF, P, DIST, NEXT, DONE, PRED)
C ***************************************************************************

C  MARCHES DOWN A SUBSTRING OF POSITIVE INTEGERS AND ONLY SORTS IF
C  ELEMENTS ARE NOT IN INCREASING ORDER.  IF A NUMBER NEEDS TO BE
C  PLACED HIGHER IN THE LIST SUBROUTINE 'PUT' IS CALLED TO DO SO

C  INPUT:  POINTER F INTO DOUBLE LINK LIST ARRAYS DIST, NEXT, AND PRED
C          TO INDICATE THE BEGINNING OF A SUBSTING OF POSITIVE INTEGERS,
C          AND PREDF

C  OUTPUT: P IS A POINTER, DIST(P) IS THE LAST POSITIVE INTEGER IN THE
C          SUBSTRING THAT IS BEGUN BY DIST(F)

      IMPLICIT INTEGER (A-Z)
      REAL*8 DIST(1000)
      LOGICAL DONE
      DIMENSION  NEXT(1000), PRED(1000)
      TAIL = F
    1 NTAIL = NEXT(TAIL)

      IF(NTAIL.EQ.0) THEN
          DONE = .TRUE.
          RETURN
      ENDIF

      IF(DIST(NTAIL).LE.DIST(TAIL)) THEN
          TAIL = NTAIL
          GOTO 1
      ENDIF

      CALL PUT2(F, TAIL, NTAIL, PREDF, DIST, NEXT, PRED)

      GOTO 1

      END
```

```
      SUBROUTINE PUT2 (F, TAIL, NTAIL, PREDF, DIST, NEXT, PRED)
C ***************************************************************************

C  REARRANGES POINTERS TO PLACE POSITIVE INTEGERS IN INCREASING ORDER.

C  INPUT:  F     -  START OF POSITIVE SUBSTRING
C          TAIL  -  END OF CURRENTLY SORTED PORTION OF SUBSTRING
C          NTAIL =  NEXT(TAIL) - POINTER TO THE SUCCESSOR OF TAIL IN THE
C                   ITEM TO BE INCORPORATED INTO THE SORTED PORTION OF
C                   THE LIST
C          PREDF -  THE PREDECESSOR OF F IN LINKED LIST

C  OUTPUT: SOME POINTERS IN NEXT AND PRED ARE CHANGED TO PUT DIST(NTAIL)
C          IN ITS PROPER PLACE IN THE SORTED PORTION ON THE LIST

      IMPLICIT INTEGER (A-Z)
      REAL*8 DIST(1000)
      DIMENSION  NEXT(1000), PRED(1000)

      IF (DIST(NTAIL).GE.DIST(F)) THEN
          TEMP = NEXT(NTAIL)
          NEXT(NTAIL) = F
          PRED(F) = NTAIL
          NEXT(TAIL) = TEMP
          PRED(TEMP) = TAIL
          F = NTAIL
          RETURN
      ENDIF
      CALL WALK2(F, TAIL, NTAIL, DIST, NEXT, PRED)
      RETURN
      END
```

```
      SUBROUTINE WALK2(F, TAIL, NTAIL, DIST, NEXT, PRED)
C ***************************************************************************

C  THIS SUBROUTINE WALKS DOWN THE LINKED LIST AND PLACES THE DIST(NTAIL)
C  IN THE CORRECT POSITION IN THE DIST ARRAY.

C  INPUT:  F, TAIL, NTAIL ARE USED AS ABOVE

C  OUTPUT: ALTERS POINTERS IN NEXT AND PRED ARRAYS TO PLACE DIST(NTAIL)
C          AFTER DIST(F) AND BEFORE DIST(TAIL) IN DOUBLE LINK LIST

      IMPLICIT INTEGER (A-Z)
      REAL*8 DIST(1000), HNTAIL
      DIMENSION  NEXT(1000), PRED(1000)

      I = F
      NEXTI = NEXT(I)
      HNTAIL = DIST(NTAIL)

    1 IF(HNTAIL.GE.DIST(NEXTI)) THEN
          NEXT(I) = NTA'L
          PRED(NTAIL) = I
          NNTAIL = NEXT(NTAIL)
          NEXT(NTAIL) = NEXTI
          PRED(NEXTI) = NTAIL
          NEXT(TAIL) = NNTAIL
          PRED(NNTAIL) = TAIL
          RETURN
      ENDIF

      I = NEXTI
      NEXTI = NEXT(I)
      GOTO 1

      END
```

```
      SUBROUTINE COUNTR(START, NEXT, HEAD, NELEMS)
C ***************************************************************************

C  THIS SUBROUTINE COUNTS THE NUMBER OF ELEMENTS IN A GIVEN STRING OF
C  CHARACTERS

C  INPUT:  HEAD() AND NEXT() ARRAYS FOR THE STRING OF CHARACTERS

C  OUTPUT:  NUMBER OF ELEMENTS IN THE STRING

      INTEGER PTR, NEXT(1000), HEAD(1000), START

      PTR = NEXT(START)
      NELEMS = 0

   20 IF (PTR.NE.0) THEN
         NELEMS = NELEMS + 1
         PTR = NEXT(PTR)
         GO TO 20
      ENDIF

      RETURN
      END
```

```
      SUBROUTINE ALG2(NOBS, NELEMS, HEAD, NEXT, X, Y, BX, BY, FR, NR,
     +                PR)
C ***************************************************************************

C  GENERATES THE WELL-KNOW FUNDAMENTAL GROUP REPRESENTATION OF AN
C  EQUIVALENCE CLASS.

C  INPUT:  RAW OR CANONICAL STRING, WITH # OBSTACLES, # ELEMENTS IN
C          STRING, COORDINATES OF A AND OBSTACLES (BK)

C  OUTPUT: 1) IF INPUT IS RAW STRING, THEN POSSIBLY UNREDUCED
C          FUNDAMENTAL GROUP WILL RESULT (IF CANCELLATION OF LIKE
C          POSITIVE NUMBERS COULD HAVE OCCURRED IN THE RAW STRING).
C          2) IF INPUT IS CANONICAL STRING, THEN THE RESULTING
C          FUNDAMENTAL GROUP WILL BE IN REDUCED FORM.

      REAL*8  XA, YA, XB, YB, X(1000), Y(1000), BX(1000), BY(1000)
      INTEGER NOBS, NELEMS, HEAD(1000), NEXT(1000), PRED(1000),
     +        FR(1000), NR(1000), PR(1000), S(1000), FUNDGP(1000),
     +        START
      LOGICAL RIGHT(1000), RITE

      LENGTH = 1
      START = NEXT(1)

      XA = X(1)
      YA = Y(1)

      DO 1 K = 1, NOBS
         XB = BX(K)
         YB = BY(K)
    1    RIGHT(K) = RITE(XA, YA, XB, YB)

      NELEM = NELEMS + 1
      DO 10 M = 2, NELEM
          S(M) = HEAD(START)
          J = IABS(S(M))
          RIGHT(J) = .NOT. RIGHT(J)
          IF(S(M).GT.0) THEN
              LENGTH = LENGTH + 1
              IF(RIGHT(J)) THEN
                   FUNDGP(LENGTH) =  J
              ELSE
                   FUNDGP(LENGTH) = -J
              ENDIF
          ENDIF
          START = NEXT(START)
          IF(START.EQ.0) GOTO 11
   10 CONTINUE
```

```
   11 FR(1) = 0
      NR(1) = 2

      DO 2 I = 2, LENGTH
          FR(I) = FUNDGP(I)
          NR(I) = I + 1
          PR(I) = I - 1
    2 CONTINUE

      NR(LENGTH) = 0

      RETURN
      END



      LOGICAL FUNCTION RITE(XA, YA, XB, YB)
C ***************************************************************************

C  THIS FUNCTION DETERMINES WHICH SIDE OF A GIVEN DIRECTED LINE ANY
C  POINT LIES.

C  INPUT: TWO POINTS THAT DETERMINE THE LINE

C  OUTPUT: LOGICAL VARIABLE THAT IS TRUE IF A POINT LIES TO THE RIGHT
C          AND FALSE IF A POINT LIES TO THE LEFT

      REAL*8  XB, YB, XA, YA, SIGNA

      RITE = .TRUE.

      SIGNA = -((YB*XA)-(XB*YA))

      IF(SIGNA.LT.0)THEN
          RITE = .FALSE.
          RETURN
      ENDIF

      IF(SIGNA.EQ.0) THEN
          PRINT*,'THE POINT A LIES ON THE LINE LK PROGRAM STOPS'
          STOP
      ENDIF

      RETURN
      END
```

```
      SUBROUTINE CANALG2(HEAD, NEXT, PRED)
C ***************************************************************************

C  THIS SUBROUTINE TAKES AN UNREDUCED FUNDAMENTAL GROUP REPRESENTATION
C  OF A GIVEN CLASS AND CANCELS A GENERATOR IF IT IS ADJACENT TO ITS
C  INVERSE.

C  INPUT:  HEAD, NEXT, AND PRED ARRAYS

C  OUTPUT: HEAD, NEXT, AND PRED ARRAYS WITH NEXT AND PRED REARRANGED
C          TO SKIP AROUND CANCELLED ELEMENTS

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      START = 1
      PTR1 = START
      PTR2 = START
      PTR3 = NEXT(PTR2)
   10 IF (PTR3.NE.0) THEN
          IF (HEAD(PTR2).EQ.-(HEAD(PTR3))) THEN
              NEXT(PTR1) = NEXT(PTR3)
              PRED(NEXT(PTR3)) = PTR1
              IF(NEXT(PTR1).EQ.0) RETURN
              PTR2 = PTR1
              PTR1 = PRED(PTR1)
              PTR3 = NEXT(PTR2)
          ELSE
              PTR1 = PTR2
              PTR2 = PTR3
              PTR3 = NEXT(PTR2)
          ENDIF
          GO TO 10
      ENDIF
      RETURN
      END
```

```
      SUBROUTINE ALG1(HEAD, NEXT, PRED, NELEMS)
C ***************************************************************************

C  THIS SUBROUTINE TAKES A GIVEN RAW STRING OF CHARACTERS REPRESENTING
C  A PATH AND PRODUCES THE CANONICAL FORM OF THAT STRING.  THIS IS DONE
C  BY FIRST ORDERING ALL OF THE ALPHA SUBSTRINGS FROM SMALLEST TO
C  LARGEST IN ABSOLUTE VALUE.  NEXT, CANCELLATION IS PERFORMED TO
C  ELIMINATE ALL LIKE PAIRS OF ADJACENT ELEMENTS FROM THE STRING.

C  INPUT:  RAW STRING IN FORM OF DOUBLE LINKED LIST WITH HEAD(), NEXT()
C          AND PRED() ARRAYS

C  OUTPUT:  CANONICAL FORM OF THE RAW STRING

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      START = 1

      CALL MERG1(HEAD, NEXT, PRED)
      CALL CANCEL(START, HEAD, NEXT, PRED, NELEMS)

      RETURN
      END
```

```
      SUBROUTINE MERG1 (HEAD, NEXT, PRED)
C ***************************************************************************

C  THIS SUBROUTINE DOES THE INITIAL SORT OF THE ALPHA SUBSTRINGS IN THE
C  RAWSTRING (INCREASING IN ABSULUTE VALUE).

C  INPUT:  HEAD, NEXT, PRED ARRAYS REPRESENTING A DOUBLE LINKED LIST
C          OF THE RAWSTRING WITH THE ALPHA SUBSTRINGS UNORDERED

C  OUTPUT: POINTERS STORED IN THE ARRAYS NEXT AND PRED ARE ALTERED SO
C          THAT EACH SUBSTRING OF THE STORED LIST WHICH CONSISTS
C          ENTIRELY OF NEGATIVE INTEGERS IS SORTED INTO NON-INCREASING
C          ORDER, WHILE SUBSTRINGS OF POSITIVE INTEGERS ARE LEFT
C          UNALTERED.

      IMPLICIT INTEGER(A-Z)
      LOGICAL DONE
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      DONE = .FALSE.
      P = 1

    1 CALL FRONT (P, F, PREDF, HEAD, NEXT, DONE)

      IF(DONE) RETURN

      CALL SORT (F, PREDF, P, HEAD, NEXT, DONE, PRED)

      IF(DONE) RETURN

      GOTO 1

      END
```

```
      SUBROUTINE FRONT(P, F, PREDF, HEAD, NEXT, DONE)
C ***************************************************************************

C  FINDS THE BEGINNING OF NEGATIVE INTEGER STRINGS (ALPHA STRING)

C  INPUT:  POINTER P INTO LINKED LIST

C  OUTPUT: F IS A POINTER INTO THE HEAD ARRAY.  POINTS TO FIRST NEGATIVE
C          ENTRY WHICH OCCURS STRICTLY AFTER HEAD(P). PREDF IS POINTER
C          SUCH THAT HEAD(PREDF) IS THE PREDESSOR OF HEAD(F).

      IMPLICIT INTEGER(A-Z)
      LOGICAL DONE
      DIMENSION HEAD(1000), NEXT(1000)

      F = P
    1 PREDF = F
      F = NEXT(F)

      IF (F.EQ.0) THEN
          DONE = .TRUE.
          RETURN
      ENDIF

      IF (HEAD(F).LT.0) RETURN

      GOTO 1

      END
```

```
      SUBROUTINE SORT(F, PREDF, P, HEAD, NEXT, DONE, PRED)
C ***************************************************************************

C  MARCHES DOWN A SUBSTRING OF NEGATIVE INTEGERS AND ONLY SORTS IF
C  ELEMENTS ARE IN INCREASING ORDER.  IF A NUMBER NEEDS TO BE
C  PLACED HIGHER IN THE LIST SUBROUTINE 'PUT' IS CALLED TO DO SO

C  INPUT:  POINTER F INTO DOUBLE LINK LIST ARRAYS HEAD, NEXT, AND PRED
C          TO INDICATE THE BEGINNING OF A SUBSTRING OF NEGATIVE
C          INTEGERS; AND PREDF

C  OUTPUT: P IS A POINTER, HEAD(P) IS THE LAST NEGATIVE INTEGER IN THE
C          SUBSTRING THAT IS BEGUN BY HEAD(F)

      IMPLICIT INTEGER (A-Z)
      LOGICAL DONE
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      TAIL = F
    1 NTAIL = NEXT(TAIL)

      IF(NTAIL.EQ.0) THEN
          DONE = .TRUE.
          RETURN
      ENDIF

      IF(HEAD(NTAIL).GT.0) THEN
          P = TAIL
          RETURN
      ENDIF

      IF(HEAD(NTAIL).LE.HEAD(TAIL)) THEN
          TAIL = NTAIL
          GOTO 1
      ENDIF

      CALL PUT(F, TAIL, NTAIL, PREDF, HEAD, NEXT, PRED)

      GOTO 1

      END
```

```
      SUBROUTINE PUT (F, TAIL, NTAIL, PREDF, HEAD, NEXT, PRED)
C ***************************************************************************

C  SUBROUTINE THAT REARRANGES POINTERS TO PLACE A NEGATIVE INTEGER IN
C  NON-INCREASING ORDER.

C  INPUT:  F     -  START OF NEGATIVE SUBSTRING
C          TAIL  -  END OF CURRENTLY SORTED PORTION OF SUBSTRING
C          NTAIL =  NEXT(TAIL) - POINTER TO THE SUCCESSOR OF TAIL IN THE
C                   ITEM TO BE INCORPORATED INTO THE SORTED PORTION OF
C                   THE LIST.
C          PREDF -  THE PREDECESSOR OF F IN LINKED LIST

C  OUTPUT: SOME POINTERS IN NEXT AND PRED ARE CHANGED TO PUT HEAD(NTAIL)
C          IN ITS PROPER PLACE IN THE SORTED PORTION ON THE LIST

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      IF (HEAD(NTAIL).GE.HEAD(F)) THEN
          NEXT(PREDF) = NTAIL
          PRED(NTAIL) = PREDF
          TEMP = NEXT(NTAIL)
          NEXT(NTAIL) = F
          PRED(F) = NTAIL
          NEXT(TAIL) = TEMP
          PRED(TEMP) = TAIL
          F = NTAIL
          RETURN
      ENDIF

      CALL WALK(F, TAIL, NTAIL, HEAD, NEXT, PRED)

      RETURN
      END
```

```
      SUBROUTINE WALK(F, TAIL, NTAIL, HEAD, NEXT, PRED)
C ***************************************************************************

C  THIS SUBROUITNE WALKS DOWN THE LINKED LIST AND PLACES THE HEAD(NTAIL)
C  IN THE CORRECT POSITION IN THE HEAD ARRAY. (DECREASING ORDER)

C  INPUT:  F, TAIL, NTAIL ARE USED AS ABOVE

C  OUTPUT: ALTERS POINTERS IN NEXT AND PRED ARRAYS TO PLACE HEAD(NTAIL)
C          AFTER HEAD(F) AND BEFORE HEAD(TAIL) IN DOUBLE LINK LIST

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      I = F
      NEXTI = NEXT(I)
      HNTAIL = HEAD(NTAIL)

    1 IF(HNTAIL.GE.HEAD(NEXTI)) THEN
          NEXT(I) = NTAIL
          PRED(NTAIL) = I
          NNTAIL = NEXT(NTAIL)
          NEXT(NTAIL) = NEXTI
          PRED(NEXTI) = NTAIL
          NEXT(TAIL) = NNTAIL
          PRED(NNTAIL) = TAIL
          RETURN
      ENDIF

      I = NEXTI
      NEXTI = NEXT(I)
      GOTO 1

      END
```

```
      SUBROUTINE CANCEL(START, HEAD, NEXT, PRED, NELEMS)
C ***************************************************************************

C  GIVEN THE SORTED RAW STRING, THIS SUBROUTINE REDUCES THE STRING TO
C  CANONICAL FORM BY ADJUSTING THE LINK LIST TO SKIP ANY PAIRS OF
C  ADJACENT LIKE ELEMENTS IN THE STRING.  ONCE TWO ELEMENTS ARE
C  ELIMINATED FROM THE STRING, THE SUBSTRINGS WHICH WERE ON THEIR LEFT
C  AND RIGHT CONCATENATE TO FORM A NEW STRING AND HENCE A NEW PAIR OF
C  ADJACENT ELEMENTS WHICH MUST ALSO BE CHECKED FOR EQUALITY.  IN THE
C  EVENT A BETA SUBSTRING BECOMES ANNHILATED FROM THE STRING, THE TWO
C  ADJACENT ALPHA STRINGS CONCATENATE AND ARE AGAIN SORTED FROM SMALLEST
C  TO LARGEST IN ABSOLUTE VALUE AS A SINGLE SUBSTRING.

C  INPUT:  SORTED RAW STRING

C  OUTPUT:  CANONICAL FORM OF THE RAW STRING

      IMPLICIT INTEGER (A-Z)
      DIMENSION HEAD(1000), NEXT(1000), PRED(1000)

      START = 1
      START1 = START
      START2 = START
      BETA = START
      PTR1 = START1
      PTR2 = START1
   50 PTR3 = NEXT(PTR2)

      IF (HEAD(PTR3).LT.0) THEN
          START1 = PTR3
          START2 = START1
      ELSE
          BETA = PTR3
      ENDIF

   10 IF (PTR3.NE.0) THEN
          CALL CHEKER (HEAD, START1, START2, PTR1, PTR2, PTR3, BETA)
          IF (HEAD(PTR2).EQ.HEAD(PTR3)) THEN
              NEXT(PTR1) = NEXT(PTR3)
              PRED(NEXT(PTR3)) = PTR1
              PTR2 = NEXT(PTR3)
              IF (PTR2.EQ.0) GO TO 60
              CHECK1 = HEAD(PTR2) * HEAD(PTR3)
              CHECK2 = HEAD(PTR1) * HEAD(PTR2)
              IF (CHECK1.LT.0.AND.CHECK2.GT.0.AND.HEAD(PTR2).LT.0)THEN
                  PTR2 = PTR1
                  PTR1 = PRED(PTR1)
                  PTR3 = NEXT(PTR2)
                  START1 = START2
    6             IF(START1.NE.1.AND.HEAD(PRED(START1)).LT.0)THEN
```

```
                START1 = PRED(START1)
                GO TO 6
            ENDIF
            START2 = PTR3
            CALL MERGE (HEAD, NEXT, START1, START2, PRED)
            PTR1 = PRED(START1)
            PTR2 = PTR1
            GO TO 50
        ELSEIF (CHECK1.LT.0.AND.CHECK2.EQ.0.AND.HEAD(PTR2).LT.0)
     +  THEN
            PTR2 = PTR1
            PTR3 = NEXT(PTR1)
            START1 = PTR3
            START2 = START1
            GO TO 10
        ELSEIF (CHECK1.GT.0.AND.CHECK2.GT.0.AND.HEAD(PTR2).GT.0)
     +  THEN
            IF (BETA.EQ.2) THEN
                BETA = 1
            ELSEIF (BETA.EQ.1) THEN
                GO TO 9
            ELSE
                IF(START1.NE.1) BETA = PRED(START1)
11              IF(HEAD(PRED(BETA)).GT.0) THEN
                    BETA = PRED(BETA)
                    GO TO 11
                ENDIF
            ENDIF
 9          PTR1 = PRED(BETA)
            PTR2 = BETA
            PTR3 = NEXT(PTR2)
            GO TO 10
        ELSEIF (CHECK1.LT.0.AND.CHECK2.GT.0.AND.HEAD(PTR2).GT.0)
     +  THEN
            PTR2 = PTR1
            PTR1 = PRED(PTR1)

            IF(HEAD(PTR1).LT.0)THEN
                START1 = PTR1
 5              IF (HEAD(PRED(START1)).LT.0)THEN
                    START1 = PRED(START1)
                    GO TO 5
                ENDIF
            ENDIF
            PTR3 = NEXT(PTR2)
            GO TO 10
        ELSEIF (CHECK1.LT.0.AND.CHECK2.LT.0.AND.HEAD(PTR2).LT.0)
     +  THEN
            IF(START1.EQ.1) THEN
                START1 = PTR2
```

```
                    START2 = PTR2
                    PTR3 = NEXT(PTR2)
                ELSE
                    START1 = START2
                    PTR3 = NEXT(PTR2)
                ENDIF
                GO TO 10
            ELSE
                START1 = START2
                PTR3 = NEXT(PTR2)
            ENDIF
        ELSE
            PTR1 = PTR2
            PTR2 = PTR3
            PTR3 = NEXT(PTR2)
        ENDIF
        GO TO 10
    ENDIF

60  CALL COUNTR (START, NEXT, HEAD, NELEMS)

    RETURN
    END
```

```
      SUBROUTINE CHEKER(HEAD, START1, START2, PTR1, PTR2, PTR3, BETA)
C ***************************************************************

C  AS POINTERS MOVE ALONG THE CHARACTER STRING DURING CANCELLATION THIS
C  SUBROUTINE CHECKS TO DETERMINE WHETHER OR NOT THE END OF ONE
C  SUBSTRING IS REACHED AND A NEW ONE BEGINS.  DEPENDING ON THE OUTCOME
C  OF THIS CHECK, THE POINTERS USED TO IDENTIFY THE BEGINNING OF THE
C  TWO LATEST ALPHA STRINGS AND THE LATEST BETA STRING ARE UPDATED

C  INPUT:  HEAD() ARRAY AND PRESENT POINTER LOCATIONS FROM 'CANCEL'
C          SUBROUTINE

C  OUTPUT:  MODIFIED INDICES FOR LOCATION OF ALPHA AND BETA STRINGS

      INTEGER HEAD(1000), START1, START2, PTR1, PTR2, PTR3, CHECK, BETA

      CHECK = HEAD(PTR3) * HEAD(PTR2)

      IF (CHECK.GT.0) THEN
          IF (HEAD(PTR2).LT.0.AND.HEAD(PTR1).GT.0) START2 = PTR2
          IF (HEAD(PTR2).GT.0.AND.HEAD(PTR1).LT.0) BETA = PTR2
      ELSEIF (CHECK.LE.0) THEN
          IF (HEAD(PTR2).GT.0.AND.HEAD(PTR1).LT.0) THEN
              BETA = PTR2
              START1 = START2
              START2 = PTR3
          ELSEIF (HEAD(PTR2).GT.0) THEN
              IF (START1.NE.1) THEN
                  START1 = START2
                  START2 = PTR3
              ELSE
                  START1 = PTR3
                  START2 = START1
              ENDIF
          ELSEIF (HEAD(PTR2).LT.0.AND.HEAD(PTR1).GT.0) THEN
              BETA = PTR3
              START1 = START2
              START2 = PTR2
          ELSEIF (HEAD(PTR2).LT.0.AND.HEAD(PTR1).LT.0) THEN
              BETA = PTR3
          ENDIF
      ENDIF

      RETURN
      END
```

```
      SUBROUTINE MERGE(HEAD, NEXT, START1, START2, PRED)
C ***************************************************************************

C  THIS SUBROUTINE MERGES TWO ORDERED ALPHA STRINGS.  THE RESULTING
C  SUBSTRING WILL BE INCREASING IN ABSOLUTE VALUE.

C  INPUT:  HEAD(), NEXT() AND PRED() ARRAYS ALONG WITH INDICES START1
C          AND START2 INDICATING BEGINNING OF THE ALPHA STRINGS TO BE
C          MERGED

C  OUTPUT:  MODIFIED STRING CONTAINING THE MERGED ALPHA STRINGS WHICH
C           WERE A RESULT OF ANNHILATION OF A BETA STRING

      INTEGER HEAD(1000), NEXT(1000), START1, START2, P1, P2, TAIL,
     +            PRED(1000)
      LOGICAL GO

      IF (START1.EQ.1) GO TO 2

      IF(HEAD(PRED(START1)).EQ.0) START1 = NEXT(PRED(START1))

      IF(NEXT(PRED(START1)).NE.START1)THEN
         START1 = NEXT(PRED(START1))
    7    IF (START1.EQ.1) GO TO 2
         IF(HEAD(START1).GT.0)THEN
             START1 = NEXT(START1)
             GO TO 7
         ENDIF
    8    IF(HEAD(PRED(START1)).LT.0)THEN
             START1 = PRED(START1)
             GO TO 8
         ENDIF
      ENDIF

    2 CALL SET(HEAD, NEXT, START1, START2, P1, P2, TAIL, PRED)

    1 IF(GO(P1, P2, START2, HEAD, NEXT)) THEN
          CALL MERGER(HEAD, NEXT, P1, P2, TAIL, PRED)
      GOTO 1
      ENDIF

      CALL FXTAIL(HEAD, NEXT, P1, P2, TAIL, START2, PRED)

      RETURN
      END
```

```
      SUBROUTINE SET(HEAD, NEXT, START1, START2, P1, P2, TAIL, PRED)
C ***************************************************************************

C  SETS THE STARTING POINTER OF THE NEW STRING AND THE TWO POINTERS
C  OF THE STRINGS TO BE MERGED

C  INPUT:  POINTERS INTO TWO ALPHA STRINGS

C  OUTPUT: POINTER INTO BEGINNING OF A NEW SORTED STRING

       INTEGER HEAD(1000), NEXT('000), START1, START2, P1, P2,TAIL,
      +        PRED(1000)

       P1 = START1
       P2 = START2
       IF(HEAD(P1).GT.HEAD(P2)) THEN
           TAIL = P1
           START1 = P1
           P1 = NEXT(P1)
      ELSE
           TAIL = P2
           START1 = P2
           NEXT(PRED(P1)) = P2
           PRED(P2) = PRED(P1)
           P2 = NEXT(P2)
       ENDIF

       RETURN
       END
```

```
      LOGICAL FUNCTION GO(P1, P2, START2, HEAD, NEXT)
C ***************************************************************************

C  LOGICAL FUNCTION TO DETERMINE WHEN THE END OF EITHER LIST IS REACHED

C  INPUT:  POINTERS INTO TWO ALPHA SUBSTRINGS

C  OUTPUT: LOGICAL VARIABLE WHICH IS TRUE IF EITHER POINTER IS AT THE
C          END OF A STRING, AND FALSE OTHERWISE

      INTEGER HEAD(1000), NEXT(1000), P1, P2, START2

      GO = .TRUE.

      IF(P1.EQ.START2)THEN
          GO = .FALSE.
          RETURN
      ENDIF

      IF(NEXT(P2).EQ.0)THEN
          GO = .FALSE.
          RETURN
      ENDIF

      IF(HEAD(P2).GT.0)THEN
          GO = .FALSE.
          RETURN
      ENDIF

      RETURN
      END
```

```
      SUBROUTINE MERGER(HEAD, NEXT, P1, P2, TAIL, PRED)
C ***************************************************************************

C  THIS SUBROUTINE COMPARES TWO ELEMENTS - ONE IN EACH ALPHA SUBSTRING
C  AND PLACES THE ONE THAT IS THE SMALLEST ON THE LIST REPRESENTING THE
C  NEW SORTED LIST

C  INPUT:  HEAD, NEXT, PRED, AND POINTERS INTO THE HEAD ARRAY

C  OUTPUT: HEAD, NEXT, PRED, AND POINTERS FOR THE NEXT TWO ELEMENTS
C          TO BE CHECKED

      INTEGER HEAD(1000), NEXT(1000), P1, P2, TAIL , PRED(1000)

      IF(HEAD(P1).GT.HEAD(P2)) THEN
         CALL TACON1(NEXT, P1, TAIL, PRED)
      ELSE
         CALL TACON2(NEXT, P2, TAIL, PRED)
      ENDIF

      RETURN
      END



      SUBROUTINE TACON1 (NEXT, P1, TAIL, PRED)
C ***************************************************************************

C  ADDS THE ELEMENT TO WHICH P1 POINTS TO THE TAIL OF THE NEW STRING

C  INPUT:  NEXT, PRED, P1, TAIL

C  OUTPUT: NEXT, PRED, P1, TAIL

      INTEGER NEXT(1000), P1, TAIL, PRED(1000)

      NEXT(TAIL) = P1
      PRED(P1) = TAIL
      TAIL = P1
      P1 = NEXT(P1)

      RETURN
      END
```

```
      SUBROUTINE TACON2 (NEXT, P2, TAIL, PRED)
C ***************************************************************************

C  ADDS THE ELEMENT TO WHICH P2 POINTS TO THE TAIL OF THE NEW STRING

C  INPUT:  NEXT, PRED, P2, TAIL

C  OUTPUT: NEXT, PRED, P2, TAIL

      INTEGER NEXT(1000), P2, TAIL, PRED(1000)

      NEXT(TAIL) = P2
      PRED(P2) = TAIL
      TAIL = P2
      P2 = NEXT(P2)
      RETURN
      END
```

```
      SUBROUTINE FXTAIL(HEAD, NEXT, P1, P2, TAIL, START2, PRED)
C ***************************************************************************

C  ATTACHES THE END OF THE LONGER SORTED STRING TO THE END OF THE MERGED
C  STRING

C  INPUT:  P1, P2, HEAD, NEXT, PRED, TAIL

C  OUTPUT: P1, P2, HEAD, NEXT, PRED, TAIL, START2

       INTEGER HEAD(1000), NEXT(1000), P1, P2, TAIL, START2, P11,
      +        PRED(1000)

       IF(P1.EQ.START2)THEN
           CALL TACON2(NEXT, P2, TAIL, PRED)
           RETURN
       ENDIF

       IF(NEXT(P2).EQ.0)THEN
           CALL TACON1(NEXT, P1, TAIL, PRED)
           P11 = TAIL
    2      IF(P1.NE.START2)THEN
               P11 = P1
               P1 = NEXT(P1)
           GOTO 2
           ENDIF
           NEXT(P11) = P2
           PRED(P2) = P11
           RETURN
       ENDIF

       IF(HEAD(P2).GT.0)THEN
           CALL TACON1(NEXT, P1, TAIL, PRED)
           P11 = TAIL
    3      IF(P1.NE.START2) THEN
               P11 = P1
               P1 = NEXT(P1)
               GOTO 3
           ENDIF
           NEXT(P11) = P2
           PRED(P2) = P11
           RETURN
       ENDIF

       RETURN
       END
```

```
      SUBROUTINE CTREX(FL, NL, PL, FR, NR, PR, BSEED, PSEED)
C ***************************************************************************

C  THIS SUBROUTINE CHECKS F(C(R(P))) AGAINST K(F(R(P))) FOR EQUALITY.

C  INPUT:  FUNDAMENTAL GROUP REPRESENTATIONS FOR BOTH ALGORITHMS

C  OUTPUT:  MESSAGE INDICATING THE OCCURRENCE OF UNEQUAL FUNDAMENTAL
C           GROUPS

      INTEGER FL(1000), NL(1000), PL(1000), FR(1000), NR(1000), PR(1000)

      IF(NL(1).EQ.0.AND.NR(1).EQ.0) GO TO 11

      I = NL(1)
      J = NR(1)

   10 IF (I.EQ.0.AND.J.EQ.0) GO TO 11

      IF (FL(I).EQ.0.AND.FR(J).EQ.0) GO TO 11

      IF (FL(I).EQ.FR(J)) THEN
          I = NL(I)
          J = NR(J)
          GOTO 10
      ELSE
          PRINT*,'STRINGS UNEQUAL'
          PRINT*,' '
          IF(NL(1).EQ.0)THEN
              PRINT*,'THE GROUP IS EMPTY AFTER ALG1 SEQUENCE'
          ELSE
              PRINT*,'THE FUNDAMENTAL GROUP AFTER ALG1 IS:'
              CALL PRINTS(FL,NL)
          ENDIF
          IF(NR(1).EQ.0) THEN
              PRINT*,'THE GROUP IS EMPTY AFTER ALG2 SEQUENCE'
          ELSE
              PRINT*,'THE FUNDAMENTAL GROUP AFTER ALG2 IS:'
              CALL PRINTS(FR,NR)
          ENDIF
          PRINT*,' '
          PRINT*,'BSEED AND PSEED ARE',BSEED,PSEED

      ENDIF

   11 RETURN
      END
```

```
      SUBROUTINE PRINTS(HEAD, NEXT)
C ***************************************************************************

C  THIS SUBROUTINE CAN BE USED TO PRINT OUT ANY STRING THAT IS STORED IN
C  LINK LIST FORM

C  INPUT:  LINK LIST ARRAYS

C  OUTPUT: HORIZONTAL STRING OF THE ELEMENTS ACCORDING TO THE NEXT()

      INTEGER HEAD(1000), NEXT(1000), STRING(1000)

      NSTART = NEXT(1)
      I = 0

   98 IF(NSTART.NE.0) THEN
          I = I + 1
          STRING(I) =  HEAD(NSTART)
          NSTART = NEXT(NSTART)
          GOTO 98
      ENDIF

      PRINT 111, (STRING(J), J = 1, I)
  111 FORMAT(' ',2014)
      PRINT*,' '

      RETURN
      END
```

```
C  ALL SUBROUTINES BELOW THIS POINT WERE USED AS DEBUG TOOLS.  THEY
C  GENERATE CRUDE GRAPHS WHICH PLOT THE BOARDS AND THE PATHS.
C  THE VALUES AQUIRED BY THE RAWSTRING , MERGE, AND MANY OTHER
C  SUBROUTINES WERE CHECKED USING THESE ROUTINES.
C  ***************************************************************************

      SUBROUTINE GRAPH(BX, BY, X, Y, NOBS, NUMPTS)
C ***************************************************************************

      REAL*8 X(1000), Y(1000), BX(1000), BY(1000)
      CHARACTER*1 MATR(53, 105)

      CALL GRAFPA(X, Y, NUMPTS, MATR)
      CALL GRAFOB(BX, BY, NOBS, MATR)

      DO 1 I = 1,53
    1    PRINT 111, (MATR(I,J), J=1,105)
  111 FORMAT(' ',105A1)

      RETURN
      END
```

```
      SUBROUTINE GRAFPA (X, Y, NUMPTS, MATR)
C ***************************************************************************

      REAL*8 X(1000), Y(1000)
      CHARACTER*1 MATR(53, 105), CH

      DO 10 L = 2,52
          DO 10 M = 2,104
   10         MATR(L,M) = ' '

      DO 1 I = 1, 105
          MATR(1,I) = '*'
    1     MATR(53,I) = '*'

      DO 4 I = 1, 53
          MATR(I,1) = '*'
    4     MATR(I,105) = '*'

      XK = X(1)
      YK = Y(1)
      CALL COOR(XK, YK, IX, IY)
      CALL CHARPA(1, CH)
      MATR(IY,IX) = CH

      DO 2 K = 2,NUMPTS
    2    CALL FILL(K, X, Y, MATR)

      DO 3 K = 2,NUMPTS
          XK = X(K)
          YK = Y(K)
          CALL COOR(XK, YK, IX, IY)
          CALL CHARPA(K, CH)
    3     MATR(IY,IX) = CH

      RETURN
      END


      SUBROUTINE COOR(BX, BY, IX, IY)
C ***************************************************************************

      REAL*8 BX, BY

      IX = MESHP(BX) * 2
      IY = 54 - MESHP(BY)

      RETURN
      END
```

```
      INTEGER FUNCTION MESHP(X)
C ***************************************************************************

      REAL*8 X

      MESHP = ((X+1.)*(51./2.)) + 2.

      RETURN
      END



      SUBROUTINE CHARPA(K, CH)
C ***************************************************************************

      CHARACTER*1 CH

      IF(K.EQ.1) THEN
          CH = 'A'
          RETURN
      END IF

      IF(K.EQ.2) THEN
          CH = 'B'
          RETURN
      END IF

      IF(K.EQ.3) THEN
          CH = 'C'
          RETURN
      ENDIF

      IF(K.EQ.4) THEN
          CH = 'D'
          RETURN
      ENDIF

      IF(K.EQ.5) THEN
          CH = 'E'
          RETURN
      ENDIF

      IF(K.EQ.6) THEN
          CH = 'A'
          RETURN
      ENDIF

      END
```

```
      SUBROUTINE FILL(K, X, Y, MATR)
C *********************************************************************
      REAL*8 X(1000), Y(1000), U, V, DX, DY, XK, YK, XKM1, YKM1
      CHARACTER*1 MATR(53,105)

      XK = X(K)
      YK = Y(K)
      XKM1 = X(K-1)
      YKM1 = Y(K-1)

      CALL COOR(XK, YK, KX, KY)

      CALL COOR(XKM1, YKM1, KXM1, KYM1)

      L = IABS(KX - KXM1) - 1
      M = IABS(KY - KYM1) - 1
      IF (M.GT.L) L = M
      DX = (XK - XKM1)/(L+1)
      DY = (YK - YKM1)/(L+1)
      DO 1 J = 1,L
          U = XKM1 + J*DX
          V = YKM1 + J*DY
          CALL COOR(U,V,IU,IV)
    1     MATR(IV, IU) = '+'

      RETURN
      END



      SUBROUTINE GRAFOB (X, Y, NOBS, MATR)
C *********************************************************************
      REAL*8 X(1000), Y(1000)
      CHARACTER*1 MATR(53, 105), CH

      MATR(27, 53) = '*'

      DO 2 K = 1,NOBS
          XK = X(K)
          YK = Y(K)
          CALL COOR(XK, YK, IX, IY)
          CALL CHAR(K, CH)
    2     MATR(IY,IX) = CH

      RETURN
      END
```

```
      SUBROUTINE CHAR(K, CH)
C ***************************************************************************
      CHARACTER*1 CH

      IF(K.EQ.1) THEN
          CH = '1'
          RETURN
      END IF

      IF(K.EQ.2) THEN
          CH = '2'
          RETURN
      END IF

      IF(K.EQ.3) THEN
          CH = '3'
          RETURN
      ENDIF

      IF(K.EQ.4) THEN
          CH = '4'
          RETURN
      ENDIF

      IF(K.EQ.5) THEN
          CH = '5'
          RETURN
      ENDIF

      IF(K.EQ.6) THEN
          CH = '6'
          RETURN
      END IF

      IF(K.EQ.7) THEN
          CH = '7'
          RETURN
      END IF

      IF(K.EQ.8) THEN
          CH = '8'
          RETURN
      ENDIF

      IF(K.EQ.9) THEN
          CH = '9'
          RETURN
      ENDIF

      IF(K.EQ.10) THEN
          CH = '@'
          RETURN
```

```
ENDIF

IF(K.EQ.11) THEN
    CH = '#'
    RETURN
END IF

IF(K.EQ.12) THEN
    CH = '$'
    RETURN
END IF

IF(K.EQ.13) THEN
    CH = '%'
    RETURN
ENDIF

IF(K.EQ.14) THEN
    CH = '&'
    RETURN
ENDIF

IF(K.EQ.15) THEN
    CH = '¢'
    RETURN
ENDIF

IF(K.EQ.16) THEN
    CH = '/'
    RETURN
END IF

IF(K.EQ.17) THEN
    CH = '\'
    RETURN
END IF

IF(K.EQ.18) THEN
    CH = '<'
    RETURN
ENDIF

IF(K.EQ.19) THEN
    CH = '>'
    RETURN
ENDIF

IF(K.EQ.20) THEN
    CH = '?'
    RETURN
ENDIF
END
```

APPENDIX B. EXAMPLE: GENERATING CLASS NAMES

This appendix illustrates the procedures used in traversing a ring graph in breadth first order to produce the names of candidate homotopy classes. The traversal is done following the rules set forth in Chapter III.

The problem considered is that of determining the names of all candidate classes of paths whose shortest representative does not cross itself between points a and z in the plane containing two fixed obstacles. Figure B.1 illustrates the reference frame for the region and the location of the points a and z.

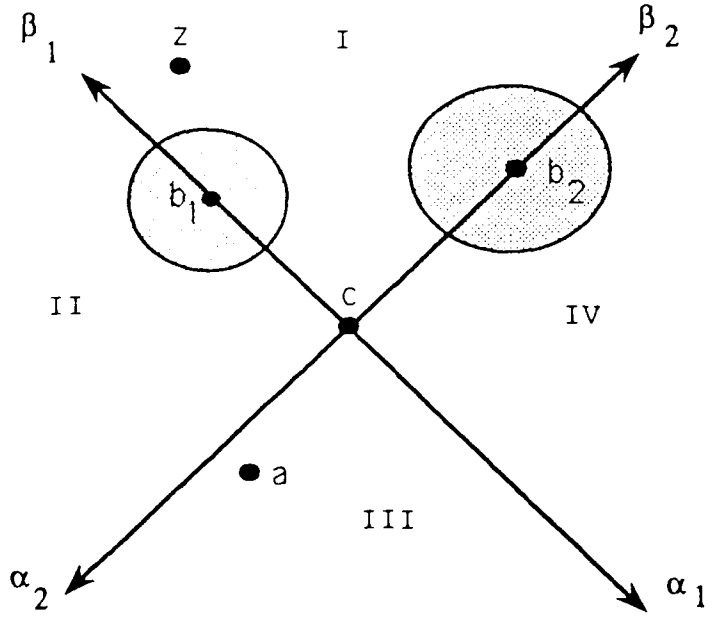


Figure B.1 Region with n = 2

Once the reference frame is established, the region is modeled with a ring graph as shown in Figure B.2. This graph is traversed beginning at node III to produce a list of all character strings representing class names and the vertex in which a path in that class ends.

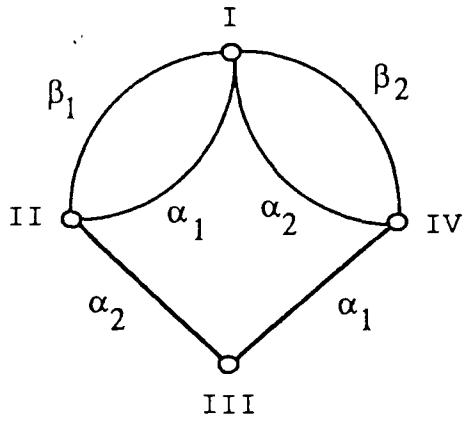


Figure B.2 Ring Graph for n = 2

For illustration purposes, as the algorithm is implemented, all k+1 long offspring of a given k long string will be listed. If a k+1 long string violates any of the "sieving" criteria presented in Chapter III, excluding consideration of the ending node, it will be marked with a (*) and the rule it violates will be stated. That string will then be withdrawn from further consideration in the algorithm. In addition to the criteria set forth in Chapter III, those strings which duplicate any previous string will also be labelled and discarded.

Upon termination of the algorithm, those strings not labelled with a (*) will be searched for strings which do not terminate at the desired node, node I. These will be labelled with an (n). Those strings not labelled with a (*) or (n) upon completion of this final check will denote the desired list of class names.

The list generated by the algorithm follows.

Violation Code	End Node	String	Violation
n	III	e	
n	II	α_2	
n	IV	α_1	
	I	$\alpha_2\beta_1$	
	I	$\alpha_2\alpha_1$	
*	I	$\alpha_1\alpha_2$	duplicates $\alpha_2\alpha_1$
	I	$\alpha_1\beta_2$	
n	II	$\alpha_2\beta_1\alpha_1$	
n	IV	$\alpha_2\beta_1\beta_2$	
n	IV	$\alpha_2\beta_1\alpha_2$	
n	II	$\alpha_2\alpha_1\beta_1$	
*	IV	$\alpha_2\alpha_1\alpha_2$	cancellation
n	IV	$\alpha_2\alpha_1\beta_2$	
n	IV	$\alpha_1\beta_2\alpha_2$	
n	II	$\alpha_1\beta_2\alpha_1$	
n	II	$\alpha_1\beta_2\beta_1$	
*	II	$\alpha_2\beta_1\alpha_1\beta_1$	wraps
n	III	$\alpha_2\beta_1\alpha_1\alpha_2$	
*	I	$\alpha_2\beta_1\beta_2\alpha_2$	wraps
n	III	$\alpha_2\beta_1\beta_2\alpha_1$	
	I	$\alpha_2\beta_1\alpha_2\beta_2$	

*	III	$\alpha_2\beta_1\alpha_2\alpha_1$	duplicates $\alpha_2\beta_1\alpha_1\alpha_2$
*	I	$\alpha_2\alpha_1\beta_1\alpha_1$	wraps
*	III	$\alpha_2\alpha_1\beta_1\alpha_2$	duplicates $\alpha_2\beta_1\alpha_1\alpha_2$
n	III	$\alpha_2\alpha_1\beta_2\alpha_1$	
*	I	$\alpha_2\alpha_1\beta_2\alpha_2$	wraps
*	III	$\alpha_1\beta_2\alpha_2\alpha_1$	duplicates $\alpha_2\alpha_1\beta_2\alpha_1$
*	I	$\alpha_1\beta_2\alpha_2\beta_2$	wraps
	I	$\alpha_1\beta_2\alpha_1\beta_1$	
*	III	$\alpha_1\beta_2\alpha_1\alpha_2$	duplicates $\alpha_2\alpha_1\beta_2\alpha_1$
*	I	$\alpha_1\beta_2\beta_1\alpha_1$	wraps
*	III	$\alpha_1\beta_2\beta_1\alpha_2$	duplicates $\alpha_2\beta_1\beta_2\alpha_1$
*	IV	$\alpha_2\beta_1\alpha_1\alpha_2\alpha_1$	cancellation
*	II	$\alpha_2\beta_1\beta_2\alpha_1\alpha_2$	wraps
*	IV	$\alpha_2\beta_1\alpha_2\beta_2\alpha_2$	wraps
*	II	$\alpha_2\beta_1\alpha_2\beta_2\alpha_1$	crosses
n	II	$\alpha_2\beta_1\alpha_2\beta_2\beta_1$	
*	II	$\alpha_2\alpha_1\beta_2\alpha_1\alpha_2$	wraps
*	II	$\alpha_1\beta_2\alpha_1\beta_1\alpha_1$	wraps
*	IV	$\alpha_1\beta_2\alpha_1\beta_1\alpha_2$	crosses
n	IV	$\alpha_1\beta_2\alpha_1\beta_1\beta_2$	
*	I	$\alpha_2\beta_1\alpha_2\beta_2\beta_1\alpha_1$	crosses
*	III	$\alpha_2\beta_1\alpha_2\beta_2\beta_1\alpha_2$	wraps
*	III	$\alpha_1\beta_2\alpha_1\beta_1\beta_2\alpha_1$	wraps
*	I	$\alpha_1\beta_2\alpha_1\beta_1\beta_2\alpha_2$	crosses

From the above list, the remaining candidate homotopy class names are:

1. $\alpha_2\beta_1$
2. $\alpha_2\alpha_1$
3. $\alpha_1\beta_2$
4. $\alpha_2\beta_1\alpha_2\beta_2$
5. $\alpha_1\beta_2\alpha_1\beta_1$.

Figures B.3 through B.7 illustrate paths which belong to each of these classes.

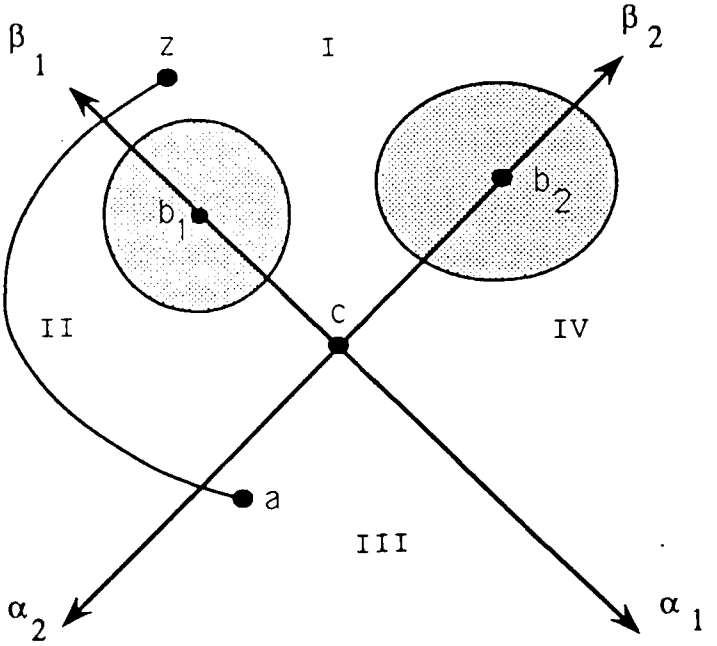


Figure B.3 Class $\alpha_1\beta_1$

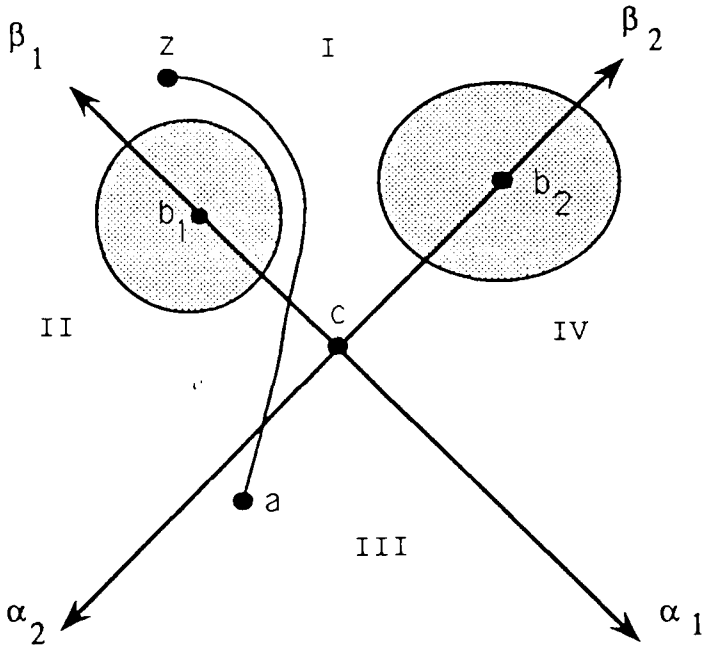


Figure B.4 Class $\alpha_2\alpha_1$

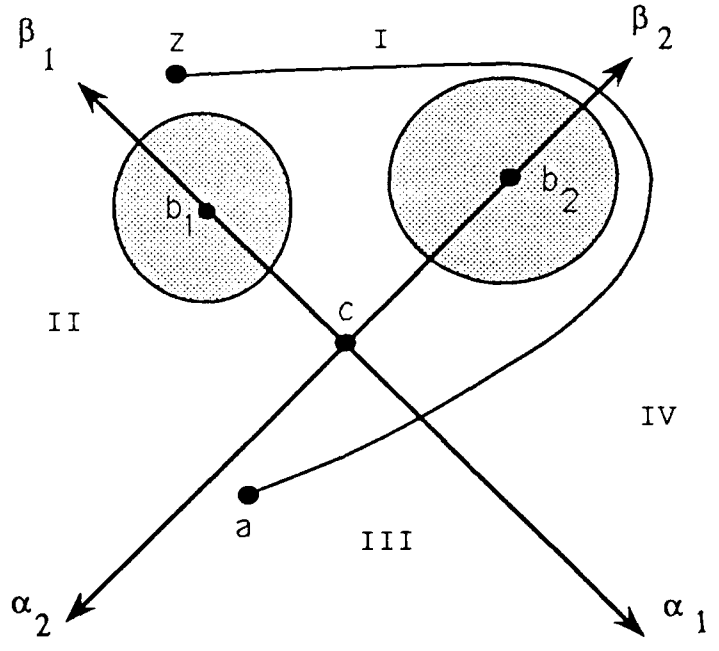


Figure B.5 Class $\alpha_1\beta_2$

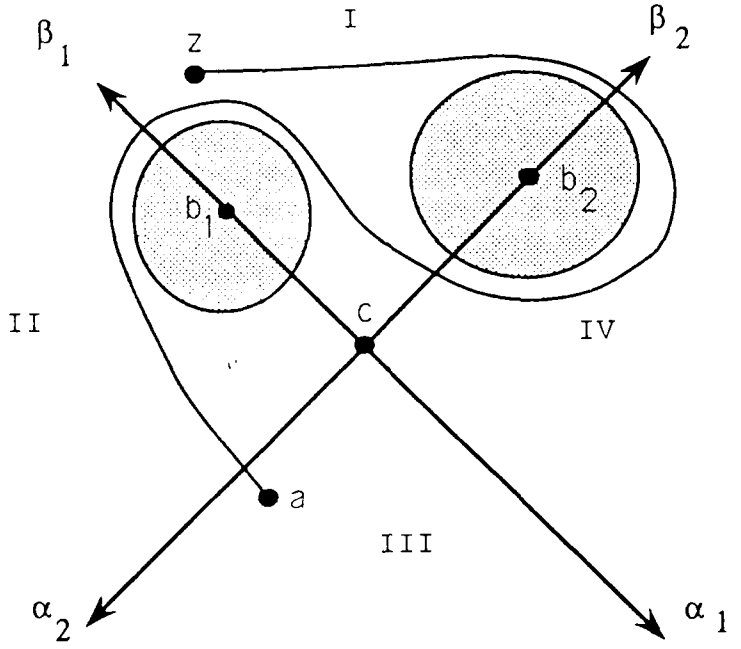


Figure B.6 Class $\alpha_2\beta_1\alpha_2\beta_2$

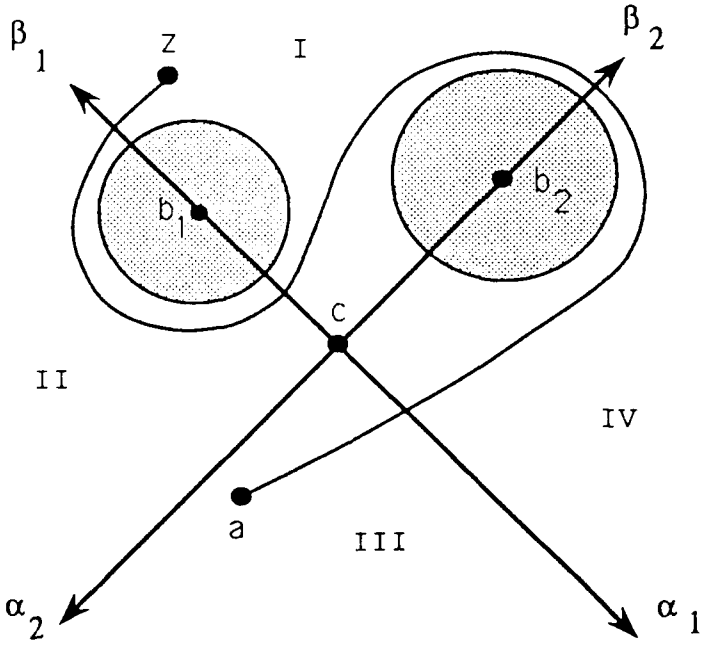


Figure B.7 Class $\alpha_1\beta_2\alpha_1\beta_1$

APPENDIX C. EXAMPLE: COUNTING NON-ISOMORPHIC GRAPHS

The material of this appendix is presented to illustrate the ideas introduced in Chapter V. The example used counts the number of non-isomorphic ring graphs given n=7 using Burnside's Lemma and then demonstrates the procedures used in determining the number of ring graphs which may be generated given a particular partition of n.

Given the number of obstacles n = 7, a regular 14-gon, as shown in Figure C.1 is produced. Edges are numbered clockwise for convenience.

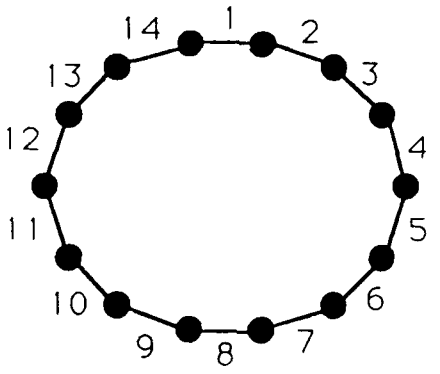


Figure C.1 14-gon resulting from n=7

The edges of the polygon are then colored black (bold) or white (fine) subject to the restriction that opposite edges are colored differently. The question to be answered is this: How many non-equivalent two-colorings exist?

The 4n = 28 total permissible permutations consist of 2n = 14 rotations of the polygon, n = 7 reflections about diagonals drawn through opposite vertices, and n = 7 reflections about bisectors drawn through opposite edges. The cycle notation for each of the 28 permutations is shown below along with the number of colorings left fixed by that permutation. These numbers are obtained by applying the formulas introduced in Chapter V. In cases where no coloring is left fixed by a permutation with more than one cycle, cycles containing opposite edges will be underlined.

Rotations

π_0 : (1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)(13)(14) $\rightarrow \Psi(\pi_0) = 2^7$

π_1 : (1 2 3 4 5 6 7 8 9 10 11 12 13 14) $\rightarrow \Psi(\pi_1) = 0$

π_2 : (1 3 5 7 9 11 13)(2 4 6 8 10 12 14) $\rightarrow \Psi(\pi_2) = 2$

π_3 : (1 4 7 10 13 2 5 8 11 14 3 6 9 12) $\rightarrow \Psi(\pi_3) = 0$

π_4 : (1 5 9 13 3 7 11)(2 6 10 14 4 8 12) $\rightarrow \Psi(\pi_4) = 2$

π_5 : (1 6 11 2 7 12 3 8 13 4 9 14 5 10) $\rightarrow \Psi(\pi_5) = 0$

π_6 : (1 7 13 5 11 3 9)(2 8 14 6 12 4 10) $\rightarrow \Psi(\pi_6) = 2$

π_7 : <u>(1 8)(2 9)(3 10)(4 11)(5 12)(6 13)(7 14)</u> $\rightarrow \Psi(\pi_7) = 0$

π_8 : (1 9 3 11 5 13 7)(2 10 4 12 6 14 8) $\rightarrow \Psi(\pi_8) = 2$

π_9 : (1 10 5 14 9 4 13 8 3 12 7 2 11 6) $\rightarrow \Psi(\pi_9) = 0$

π_{10} : (1 11 7 3 13 9 5)(2 12 8 4 14 10 6) $\rightarrow \Psi(\pi_{10}) = 2$

π_{11} : (1 12 9 6 3 14 11 8 5 2 13 10 7 4) $\rightarrow \Psi(\pi_{11}) = 0$

π_{12} : (1 13 11 9 7 5 3)(2 14 12 10 8 6 4) $\rightarrow \Psi(\pi_{12}) = 2$

π_{13} : (1 14 13 12 11 10 9 8 7 6 5 4 3 2) $\rightarrow \Psi(\pi_{13}) = 0$

Reflections About Diagonals

π_{14} : (1 14)(2 13)(3 12)(4 11)(5 10)(6 9)(7 8) $\rightarrow \Psi(\pi_{14}) = 0$

π_{15} : (1 2)(3 14)(4 13)(5 12)(6 11)(7 10)(8 9) $\rightarrow \Psi(\pi_{15}) = 0$

π_{16} : (2 3)(1 4)(5 14)(6 13)(7 12)(8 11)(9 10) $\rightarrow \Psi(\pi_{16}) = 0$

π_{17} : (3 4)(2 5)(1 6)(7 14)(8 13)(9 12)(10 11) $\rightarrow \Psi(\pi_{17}) = 0$

π_{18} : (4 5)(3 6)(2 7)(1 8)(9 14)(10 13)(11 12) $\rightarrow \Psi(\pi_{18}) = 0$

π_{19} : (5 6)(4 7)(3 8)(2 9)(1 10)(11 14)(12 13) $\rightarrow \Psi(\pi_{19}) = 0$

π_{20} : (6 7)(5 8)(4 9)(3 10)(2 11)(1 12)(13 14) $\rightarrow \Psi(\pi_{20}) = 0$

Reflections About Bisectors

π_{21} : (1)(8)(2 14)(3 13)(4 12)(5 11)(6 10)(7 9) $\rightarrow \Psi(\pi_{21}) = 2^4$

π_{22} : (2)(9)(1 3)(4 14)(5 13)(6 12)(7 11)(8 10) $\rightarrow \Psi(\pi_{22}) = 2^4$

π_{23} : (3)(10)(2 4)(1 5)(6 14)(7 13)(8 12)(9 11) $\rightarrow \Psi(\pi_{23}) = 2^4$

π_{24} : (4)(11)(3 5)(2 6)(1 7)(8 14)(9 13)(10 12) $\rightarrow \Psi(\pi_{24}) = 2^4$

π_{25} : (5)(12)(4 6)(3 7)(2 8)(1 9)(10 14)(11 13) $\rightarrow \Psi(\pi_{25}) = 2^4$

π_{26} : (6)(13)(5 7)(4 8)(3 9)(2 10)(1 11)(12 14) $\rightarrow \Psi(\pi_{26}) = 2^4$

π_{27} : (7)(14)(6 8)(5 9)(4 10)(3 11)(2 12)(1 13) $\rightarrow \Psi(\pi_{27}) = 2^4$

Applying Burnside's Lemma to the above permutations gives

$$N = (1/28)[2^7+2+2+2+2+2+2+2^4+2^4+2^4+2^4+2^4+2^4+2^4]$$

or, N = 9.

Thus, the number of non-equivalent edge two-colorings of the polygon, and hence the number of non-isomorphic ring graphs for n = 7 fixed obstacles is obtained.

Next, we consider the partitions of the integer 7. They are given below with an arrow (→) indicating those with an odd number of summands.

```
→ 7
  6 1
  5 2
→ 5 1 1
  4 3
→ 4 2 1
  4 1 1 1
→ 3 3 1
→ 3 2 2
  3 2 1 1
→ 3 1 1 1 1
  2 2 2 1
→ 2 2 1 1 1
  2 1 1 1 1 1
→ 1 1 1 1 1 1 1
```

Consider the partition 2,2,1,1,1 with k = 5 summands. Applying the information from Chapter V, a pentagon (since k = 5) is constructed as shown in Figure C.2. Vertices are numbered clockwise for convenience.

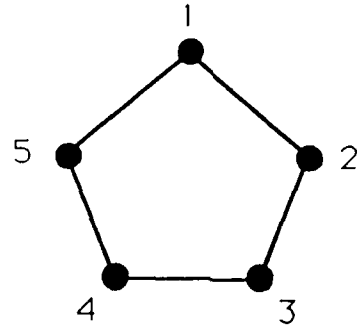


Figure C.2 Pentagon Corresponding to Partition where k = 5

The $2k = 10$ permutations of the pentagon consist of five rotations and five reflections about axes drawn through a vertex and bisecting its opposite edge. Each permutation is listed below, along with its cycle notation and cycle structure representation.

Rotations

π_0 : $(1)(2)(3)(4)(5) \rightarrow x_1^5$

π_1 : $(1\ 2\ 3\ 4\ 5) \rightarrow x_5^1$

π_2 : $(1\ 3\ 5\ 2\ 4) \rightarrow x_5^1$

π_3 : $(1\ 4\ 2\ 5\ 3) \rightarrow x_5^1$

π_4 : $(1\ 5\ 4\ 3\ 2) \rightarrow x_5^1$

Reflections

π_5 : $(1)(2\ 5)(3\ 4) \rightarrow x_1^1\ x_2^2$

π_6 : $(1\ 3)(2)(4\ 5) \rightarrow x_1^1\ x_2^2$

π_7 : $(1\ 5)(2\ 4)(3) \rightarrow x_1^1\ x_2^2$

π_8 : $(1\ 2)(3\ 5)(4) \rightarrow x_1^1\ x_2^2$

π_9 : $(1\ 4)(2\ 3)(5) \rightarrow x_1^1\ x_2^2$

These permutations result in the cycle index

$$P_G = \frac{1}{10}(x_1^5 + 4x_5^1 + 5x_1^1x_2^2) \ .$$

In applying Polya's Theorem to determine the number of non-isomorphic ring graphs generated by this partition, the

number of colors used is two, since the partition consists only of the summands 2 and 1. Letting these colors be b and w for black and white respectively,

$$P_G((b+w),\ (b^2+w^2),\ldots,\ (b^5+w^5))$$

yields

$$\frac{1}{10}\left[(b+w)^5 + 4(b^5+w^5) + 5(b+w)(b^2+w^2)^2\right] .$$

From this index, the coefficient of b^2w^3 is desired since the summand 2 appears twice in the partition and the summand 1 appears three times.

Expanding the index gives

$$\frac{1}{10}\left[10b^5 + 10b^4w + 20b^3w^2 + 20b^2w^3 + 10bw^4 + 10w^5\right] .$$

Hence, the coefficient of b^2w^3 is 2, which <u>is</u> the number of non-equivalent two-colorings which may be produced from the partition 2,2,1,1,1. The two corresponding non-isomorphic ring graphs are shown in Figure C.3. This same procedure is applied to the remaining candidate partitions to find the number of non-isomorphic ring graphs which may be produced from each one.

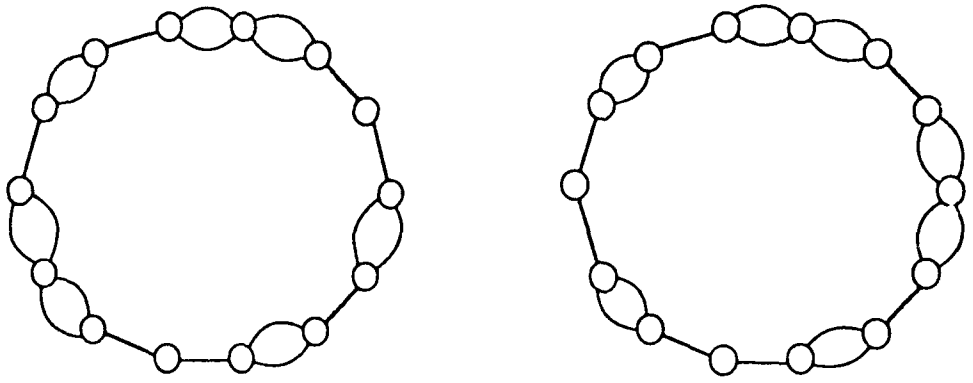


Figure C.3 Non-isomorphic Ring Graphs Produced Using Summands 2,2,1,1,1

LIST OF REFERENCES

1. Willard, S., General Topology, Addison-Wesley Publishing Company, 1970.

2. Thornton, J.R., *Algebraic Names for Homotopy Classes of Paths in a Plane With Obstacles: A Foundation for the Shortest Path Problem*, NPS Technical Report, Unpublished.

3. Cuerington, A.M., *The Shortest Path Problem in the Plane With Obstacles: Bounds on Path Lengths and Shortest Paths Within Homotopy Classes*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1991.

4. Carrol, J., and Long, D., *Theory of Finite Automata*, Prentice-Hall, Inc., 1989.

5. Dossey, J.A., and others, *Discrete Mathematics*, Scott, Foresman and Company, 1987.

6. Buckley, F., and Harary, F., *Distance in Graphs*, Addison-Wesley Publishing Company, 1990.

7. Tucker, A., *Applied Combinatorics*, 2d ed., John Wiley & Sons, Inc., 1984.

8. Hillman, A.P., and Alexandersen, G.L., *A First Undergraduate Course in Abstract Algebra*, 4th ed., Wadsworth Publishing Company, 1988.

9. Fredricksen, H.M., and Kessler, I.J., "An Algorithm For Generating Necklaces of Beads in Two Colors," *Discrete Mathematics*, v. 61, pp. 181-188, 1986.

BIBLIOGRAPHY

Patterson, E.M., *Topology*, 2d ed., Interscience Publishers Inc., 1959.

Read, R.C., "Polya's Theorem and its Progeny," *Mathematics Magazine*, v. 60, no. 5, pp. 275-282, December 1987.

Roberts, F.S., *Applied Combinatorics*, Prentice-Hall, Inc., 1984.

Simmons, G.F., *Introduction to Topology and Modern Analysis*, McGraw-Hill Book Company, Inc., 1963.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	1
4.	Professor John Thornton, Code MA Th Department of Mathematics Naval Postgraduate School Monterey, California 93943-5000	2
5.	Chairman, Code MA Department of Mathematics Naval Postgraduate School Monterey, California 93943-5000	1
6.	Professor Kim Query, Code MA Hk Department of Mathematics Naval Postgraduate School Monterey, California 93943-5000	1
7.	Captain Kevin D. Jenkins, USMC 712 Lisburn Road Camp Hill, Pennsylvania 17011	3
8.	Captain André M. Cuerington, USA P.O. Box 114 Highland Falls, New York 10928	1
9.	Commandant of the Marine Corps Code MA Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001 Attn: Captain D.W. Hintze	1